

Xen and the Art of Virtualization

Paul Barham*, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris,
Alex Ho, Rolf Neugebauer[†], Ian Pratt, Andrew Warfield

University of Cambridge Computer Laboratory
15 JJ Thomson Avenue, Cambridge, UK, CB3 0FD
{firstname.lastname}@cl.cam.ac.uk

ABSTRACT

Numerous systems have been designed which use virtualization to subdivide the ample resources of a modern computer. Some require specialized hardware, or cannot support commodity operating systems. Some target 100% binary compatibility at the expense of performance. Others sacrifice security or functionality for speed. Few offer resource isolation or performance guarantees; most provide only best-effort provisioning, risking denial of service.

This paper presents Xen, an x86 virtual machine monitor which allows multiple commodity operating systems to share conventional hardware in a safe and resource managed fashion, but without sacrificing either performance or functionality. This is achieved by providing an idealized virtual machine abstraction to which operating systems such as Linux, BSD and Windows XP, can be *ported* with minimal effort.

Our design is targeted at hosting up to 100 virtual machine instances simultaneously on a modern server. The virtualization approach taken by Xen is extremely efficient: we allow operating systems such as Linux and Windows XP to be hosted simultaneously for a negligible performance overhead — at most a few percent compared with the unvirtualized case. We considerably outperform competing commercial and freely available solutions in a range of microbenchmarks and system-wide tests.

Categories and Subject Descriptors

D.4.1 [Operating Systems]: Process Management; D.4.2 [Operating Systems]: Storage Management; D.4.8 [Operating Systems]: Performance

General Terms

Design, Measurement, Performance

Keywords

Virtual Machine Monitors, Hypervisors, Paravirtualization

*Microsoft Research Cambridge, UK

[†]Intel Research Cambridge, UK

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

Copyright 2003 ACM 1-58113-757-5/03/0010 ...\$5.00.

1. INTRODUCTION

Modern computers are sufficiently powerful to use virtualization to present the illusion of many smaller *virtual machines* (VMs), each running a separate operating system instance. This has led to a resurgence of interest in VM technology. In this paper we present Xen, a high performance resource-managed virtual machine monitor (VMM) which enables applications such as server consolidation [42, 8], co-located hosting facilities [14], distributed web services [43], secure computing platforms [12, 16] and application mobility [26, 37].

Successful partitioning of a machine to support the concurrent execution of multiple operating systems poses several challenges. Firstly, virtual machines must be isolated from one another; it is not acceptable for the execution of one to adversely affect the performance of another. This is particularly true when virtual machines are owned by mutually untrusting users. Secondly, it is necessary to support a variety of different operating systems to accommodate the heterogeneity of popular applications. Thirdly, the performance overhead introduced by virtualization should be small.

Xen hosts commodity operating systems, albeit with some source modifications. The prototype described and evaluated in this paper can support multiple concurrent instances of our XenoLinux guest operating system; each instance exports an application binary interface identical to a non-virtualized Linux 2.4. Our port of Windows XP to Xen is not yet complete but is capable of running simple user-space processes. Work is also progressing in porting NetBSD.

Xen enables users to dynamically instantiate an operating system to execute whatever they desire. In the XenoServer project [15, 35] we are deploying Xen on standard server hardware at economically strategic locations within ISPs or at Internet exchanges. We perform admission control when starting new virtual machines and expect each VM to *pay* in some fashion for the resources it requires. We discuss our ideas and approach in this direction elsewhere [21]; this paper focuses on the VMM.

There are a number of ways to build a system to host multiple applications and servers on a shared machine. Perhaps the simplest is to deploy one or more hosts running a standard operating system such as Linux or Windows, and then to allow users to install files and start processes — protection between applications being provided by conventional OS techniques. Experience shows that system administration can quickly become a time-consuming task due to complex configuration interactions between supposedly disjoint applications.

More importantly, such systems do not adequately support performance isolation; the scheduling priority, memory demand, network traffic and disk accesses of one process impact the performance of others. This may be acceptable when there is adequate provisioning and a closed user group (such as in the case of com-

putational grids, or the experimental PlanetLab platform [33]), but not when resources are oversubscribed, or users uncooperative.

One way to address this problem is to retrofit support for performance isolation to the operating system. This has been demonstrated to a greater or lesser degree with resource containers [3], Linux/RK [32], QLinux [40] and SILK [4]. One difficulty with such approaches is ensuring that *all* resource usage is accounted to the correct process — consider, for example, the complex interactions between applications due to buffer cache or page replacement algorithms. This is effectively the problem of “QoS crosstalk” [41] within the operating system. Performing multiplexing at a low level can mitigate this problem, as demonstrated by the Exokernel [23] and Nemesis [27] operating systems. Unintentional or undesired interactions between tasks are minimized.

We use this same basic approach to build Xen, which multiplexes physical resources at the granularity of an entire operating system and is able to provide performance isolation between them. In contrast to process-level multiplexing this also allows a range of guest operating systems to gracefully coexist rather than mandating a specific application binary interface. There is a price to pay for this flexibility — running a full OS is more heavyweight than running a process, both in terms of initialization (e.g. booting or resuming versus fork and exec), and in terms of resource consumption.

For our target of up to 100 hosted OS instances, we believe this price is worth paying; it allows individual users to run unmodified binaries, or collections of binaries, in a resource controlled fashion (for instance an Apache server along with a PostgreSQL backend). Furthermore it provides an extremely high level of flexibility since the user can dynamically create the precise execution environment their software requires. Unfortunate configuration interactions between various services and applications are avoided (for example, each Windows instance maintains its own registry).

The remainder of this paper is structured as follows: in Section 2 we explain our approach towards virtualization and outline how Xen works. Section 3 describes key aspects of our design and implementation. Section 4 uses industry standard benchmarks to evaluate the performance of XenoLinux running above Xen in comparison with stand-alone Linux, VMware Workstation and User-mode Linux (UML). Section 5 reviews related work, and finally Section 6 discusses future work and concludes.

2. XEN: APPROACH & OVERVIEW

In a traditional VMM the virtual hardware exposed is functionally identical to the underlying machine [38]. Although *full virtualization* has the obvious benefit of allowing unmodified operating systems to be hosted, it also has a number of drawbacks. This is particularly true for the prevalent IA-32, or x86, architecture.

Support for full virtualization was never part of the x86 architectural design. Certain supervisor instructions must be handled by the VMM for correct virtualization, but executing these with insufficient privilege fails silently rather than causing a convenient trap [36]. Efficiently virtualizing the x86 MMU is also difficult. These problems can be solved, but only at the cost of increased complexity and reduced performance. VMware’s ESX Server [10] dynamically rewrites portions of the hosted machine code to insert traps wherever VMM intervention might be required. This translation is applied to the entire guest OS kernel (with associated translation, execution, and caching costs) since all non-trapping privileged instructions must be caught and handled. ESX Server implements shadow versions of system structures such as page tables and maintains consistency with the virtual tables by trapping every update attempt — this approach has a high cost for update-intensive operations such as creating a new application process.

Notwithstanding the intricacies of the x86, there are other arguments against full virtualization. In particular, there are situations in which it is desirable for the hosted operating systems to see real as well as virtual resources: providing both real and virtual time allows a guest OS to better support time-sensitive tasks, and to correctly handle TCP timeouts and RTT estimates, while exposing real machine addresses allows a guest OS to improve performance by using superpages [30] or page coloring [24].

We avoid the drawbacks of full virtualization by presenting a virtual machine abstraction that is similar but not identical to the underlying hardware — an approach which has been dubbed *paravirtualization* [43]. This promises improved performance, although it does require modifications to the guest operating system. It is important to note, however, that we do not require changes to the application binary interface (ABI), and hence no modifications are required to guest *applications*.

We distill the discussion so far into a set of design principles:

1. Support for unmodified application binaries is essential, or users will not transition to Xen. Hence we must virtualize all architectural features required by existing standard ABIs.
2. Supporting full multi-application operating systems is important, as this allows complex server configurations to be virtualized within a single guest OS instance.
3. Paravirtualization is necessary to obtain high performance and strong resource isolation on uncooperative machine architectures such as x86.
4. Even on cooperative machine architectures, completely hiding the effects of resource virtualization from guest OSes risks both correctness and performance.

Note that our paravirtualized x86 abstraction is quite different from that proposed by the recent Denali project [44]. Denali is designed to support thousands of virtual machines running network services, the vast majority of which are small-scale and unpopular. In contrast, Xen is intended to scale to approximately 100 virtual machines running industry standard applications and services. Given these very different goals, it is instructive to contrast Denali’s design choices with our own principles.

Firstly, Denali does not target existing ABIs, and so can elide certain architectural features from their VM interface. For example, Denali does not fully support x86 segmentation although it is exported (and widely used¹) in the ABIs of NetBSD, Linux, and Windows XP.

Secondly, the Denali implementation does not address the problem of supporting application multiplexing, nor multiple address spaces, within a single guest OS. Rather, applications are linked explicitly against an instance of the Ilwaco guest OS in a manner rather reminiscent of a libOS in the Exokernel [23]. Hence each virtual machine essentially hosts a single-user single-application unprotected “operating system”. In Xen, by contrast, a single virtual machine hosts a real operating system which may itself securely multiplex thousands of unmodified user-level processes. Although a prototype *virtual MMU* has been developed which may help Denali in this area [44], we are unaware of any published technical details or evaluation.

Thirdly, in the Denali architecture the VMM performs all paging to and from disk. This is perhaps related to the lack of memory-management support at the virtualization layer. Paging within the

¹For example, segments are frequently used by thread libraries to address thread-local data.

Memory Management Segmentation	Cannot install fully-privileged segment descriptors and cannot overlap with the top end of the linear address space.
Paging	Guest OS has direct read access to hardware page tables, but updates are batched and validated by the hypervisor. A domain may be allocated discontinuous machine pages.
CPU	
Protection	Guest OS must run at a lower privilege level than Xen.
Exceptions	Guest OS must register a descriptor table for exception handlers with Xen. Aside from page faults, the handlers remain the same.
System Calls	Guest OS may install a 'fast' handler for system calls, allowing direct calls from an application into its guest OS and avoiding indirecting through Xen on every call.
Interrupts	Hardware interrupts are replaced with a lightweight event system.
Time	Each guest OS has a timer interface and is aware of both 'real' and 'virtual' time.
Device I/O	
Network, Disk, etc.	Virtual devices are elegant and simple to access. Data is transferred using asynchronous I/O rings. An event mechanism replaces hardware interrupts for notifications.

Table 1: The paravirtualized x86 interface.

VMM is contrary to our goal of performance isolation: malicious virtual machines can encourage thrashing behaviour, unfairly depriving others of CPU time and disk bandwidth. In Xen we expect each guest OS to perform its own paging using its own guaranteed memory reservation and disk allocation (an idea previously exploited by self-paging [20]).

Finally, Denali virtualizes the 'namespaces' of all machine resources, taking the view that no VM can access the resource allocations of another VM if it cannot name them (for example, VMs have no knowledge of hardware addresses, only the virtual addresses created for them by Denali). In contrast, we believe that secure access control within the hypervisor is sufficient to ensure protection; furthermore, as discussed previously, there are strong correctness and performance arguments for making physical resources directly visible to guest OSes.

In the following section we describe the virtual machine abstraction exported by Xen and discuss how a guest OS must be modified to conform to this. Note that in this paper we reserve the term *guest operating system* to refer to one of the OSes that Xen can host and we use the term *domain* to refer to a running virtual machine within which a guest OS executes; the distinction is analogous to that between a *program* and a *process* in a conventional system. We call Xen itself the *hypervisor* since it operates at a higher privilege level than the supervisor code of the guest operating systems that it hosts.

2.1 The Virtual Machine Interface

Table 1 presents an overview of the paravirtualized x86 interface, factored into three broad aspects of the system: memory management, the CPU, and device I/O. In the following we address each machine subsystem in turn, and discuss how each is presented in our paravirtualized architecture. Note that although certain parts of our implementation, such as memory management, are specific to the x86, many aspects (such as our virtual CPU and I/O devices) can be readily applied to other machine architectures. Furthermore, x86 represents a *worst case* in the areas where it differs significantly from RISC-style processors — for example, efficiently virtualizing hardware page tables is more difficult than virtualizing a software-managed TLB.

2.1.1 Memory management

Virtualizing memory is undoubtedly the most difficult part of paravirtualizing an architecture, both in terms of the mechanisms required in the hypervisor and modifications required to port each

guest OS. The task is easier if the architecture provides a software-managed TLB as these can be efficiently virtualized in a simple manner [13]. A tagged TLB is another useful feature supported by most server-class RISC architectures, including Alpha, MIPS and SPARC. Associating an address-space identifier tag with each TLB entry allows the hypervisor and each guest OS to efficiently coexist in separate address spaces because there is no need to flush the entire TLB when transferring execution.

Unfortunately, x86 does not have a software-managed TLB; instead TLB misses are serviced automatically by the processor by walking the page table structure in hardware. Thus to achieve the best possible performance, all valid page translations for the current address space should be present in the hardware-accessible page table. Moreover, because the TLB is not tagged, address space switches typically require a complete TLB flush. Given these limitations, we made two decisions: (i) guest OSes are responsible for allocating and managing the hardware page tables, with minimal involvement from Xen to ensure safety and isolation; and (ii) Xen exists in a 64MB section at the top of every address space, thus avoiding a TLB flush when entering and leaving the hypervisor.

Each time a guest OS requires a new page table, perhaps because a new process is being created, it allocates and initializes a page from its own memory reservation and registers it with Xen. At this point the OS must relinquish direct write privileges to the page-table memory: all subsequent updates must be validated by Xen. This restricts updates in a number of ways, including only allowing an OS to map pages that it owns, and disallowing writable mappings of page tables. Guest OSes may *batch* update requests to amortize the overhead of entering the hypervisor. The top 64MB region of each address space, which is reserved for Xen, is not accessible or remappable by guest OSes. This address region is not used by any of the common x86 ABIs however, so this restriction does not break application compatibility.

Segmentation is virtualized in a similar way, by validating updates to hardware segment descriptor tables. The only restrictions on x86 segment descriptors are: (i) they must have lower privilege than Xen, and (ii) they may not allow any access to the Xen-reserved portion of the address space.

2.1.2 CPU

Virtualizing the CPU has several implications for guest OSes. Principally, the insertion of a hypervisor below the operating system violates the usual assumption that the OS is the most privileged

entity in the system. In order to protect the hypervisor from OS misbehavior (and domains from one another) guest OSes must be modified to run at a lower privilege level.

Many processor architectures only provide two privilege levels. In these cases the guest OS would share the lower privilege level with applications. The guest OS would then protect itself by running in a separate address space from its applications, and indirectly pass control to and from applications via the hypervisor to set the virtual privilege level and change the current address space. Again, if the processor's TLB supports address-space tags then expensive TLB flushes can be avoided.

Efficient virtualization of privilege levels is possible on x86 because it supports four distinct privilege levels in hardware. The x86 privilege levels are generally described as *rings*, and are numbered from zero (most privileged) to three (least privileged). OS code typically executes in ring 0 because no other ring can execute privileged instructions, while ring 3 is generally used for application code. To our knowledge, rings 1 and 2 have not been used by any well-known x86 OS since OS/2. Any OS which follows this common arrangement can be ported to Xen by modifying it to execute in ring 1. This prevents the guest OS from directly executing privileged instructions, yet it remains safely isolated from applications running in ring 3.

Privileged instructions are paravirtualized by requiring them to be validated and executed within Xen— this applies to operations such as installing a new page table, or yielding the processor when idle (rather than attempting to `hlt` it). Any guest OS attempt to directly execute a privileged instruction is failed by the processor, either silently or by taking a fault, since only Xen executes at a sufficiently privileged level.

Exceptions, including memory faults and software traps, are virtualized on x86 very straightforwardly. A table describing the handler for each type of exception is registered with Xen for validation. The handlers specified in this table are generally identical to those for real x86 hardware; this is possible because the exception stack frames are unmodified in our paravirtualized architecture. The sole modification is to the page fault handler, which would normally read the faulting address from a privileged processor register (CR2); since this is not possible, we write it into an extended stack frame². When an exception occurs while executing outside ring 0, Xen's handler creates a copy of the exception stack frame on the guest OS stack and returns control to the appropriate registered handler.

Typically only two types of exception occur frequently enough to affect system performance: system calls (which are usually implemented via a software exception), and page faults. We improve the performance of system calls by allowing each guest OS to register a 'fast' exception handler which is accessed directly by the processor without indirecting via ring 0; this handler is validated before installing it in the hardware exception table. Unfortunately it is not possible to apply the same technique to the page fault handler because only code executing in ring 0 can read the faulting address from register CR2; page faults must therefore always be delivered via Xen so that this register value can be saved for access in ring 1.

Safety is ensured by validating exception handlers when they are presented to Xen. The only required check is that the handler's code segment does not specify execution in ring 0. Since no guest OS can create such a segment, it suffices to compare the specified segment selector to a small number of static values which are reserved by Xen. Apart from this, any other handler problems are fixed up during exception propagation — for example, if the handler's code

²In hindsight, writing the value into a pre-agreed shared memory location rather than modifying the stack frame would have simplified the XP port.

OS subsection	# lines	
	Linux	XP
Architecture-independent	78	1299
Virtual network driver	484	—
Virtual block-device driver	1070	—
Xen-specific (non-driver)	1363	3321
Total	2995	4620
(Portion of total x86 code base	1.36%	0.04%)

Table 2: The simplicity of porting commodity OSes to Xen. The cost metric is the number of lines of reasonably commented and formatted code which are modified or added compared with the original x86 code base (excluding device drivers).

segment is not present or if the handler is not paged into memory then an appropriate fault will be taken when Xen executes the `iret` instruction which returns to the handler. Xen detects these "double faults" by checking the faulting program counter value: if the address resides within the exception-virtualizing code then the offending guest OS is terminated.

Note that this "lazy" checking is safe even for the direct system-call handler: access faults will occur when the CPU attempts to directly jump to the guest OS handler. In this case the faulting address will be outside Xen (since Xen will never execute a guest OS system call) and so the fault is virtualized in the normal way. If propagation of the fault causes a further "double fault" then the guest OS is terminated as described above.

2.1.3 Device I/O

Rather than emulating existing hardware devices, as is typically done in fully-virtualized environments, Xen exposes a set of clean and simple device abstractions. This allows us to design an interface that is both efficient and satisfies our requirements for protection and isolation. To this end, I/O data is transferred to and from each domain via Xen, using shared-memory, asynchronous buffer-descriptor rings. These provide a high-performance communication mechanism for passing buffer information vertically through the system, while allowing Xen to efficiently perform validation checks (for example, checking that buffers are contained within a domain's memory reservation).

Similar to hardware interrupts, Xen supports a lightweight event-delivery mechanism which is used for sending asynchronous notifications to a domain. These notifications are made by updating a bitmap of pending event types and, optionally, by calling an event handler specified by the guest OS. These callbacks can be 'held off' at the discretion of the guest OS — to avoid extra costs incurred by frequent wake-up notifications, for example.

2.2 The Cost of Porting an OS to Xen

Table 2 demonstrates the cost, in lines of code, of porting commodity operating systems to Xen's paravirtualized x86 environment. Note that our NetBSD port is at a very early stage, and hence we report no figures here. The XP port is more advanced, but still in progress; it can execute a number of user-space applications from a RAM disk, but it currently lacks any virtual I/O drivers. For this reason, figures for XP's virtual device drivers are not presented. However, as with Linux, we expect these drivers to be small and simple due to the idealized hardware abstraction presented by Xen.

Windows XP required a surprising number of modifications to its architecture independent OS code because it uses a variety of structures and unions for accessing page-table entries (PTEs). Each page-table access had to be separately modified, although some of

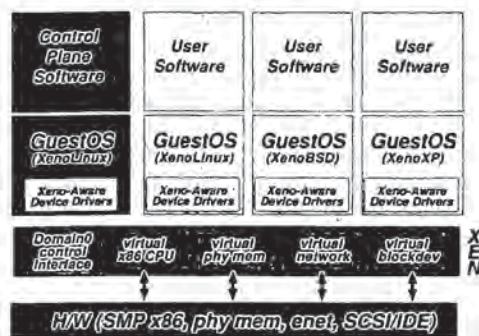


Figure 1: The structure of a machine running the Xen hypervisor, hosting a number of different guest operating systems, including *Domain0* running control software in a XenoLinux environment.

this process was automated with scripts. In contrast, Linux needed far fewer modifications to its generic memory system as it uses pre-processor macros to access PTEs — the macro definitions provide a convenient place to add the translation and hypervisor calls required by paravirtualization.

In both OSes, the architecture-specific sections are effectively a port of the x86 code to our paravirtualized architecture. This involved rewriting routines which used privileged instructions, and removing a large amount of low-level system initialization code. Again, more changes were required in Windows XP, mainly due to the presence of legacy 16-bit emulation code and the need for a somewhat different boot-loading mechanism. Note that the x86-specific code base in XP is substantially larger than in Linux and hence a larger porting effort should be expected.

2.3 Control and Management

Throughout the design and implementation of Xen, a goal has been to separate policy from mechanism wherever possible. Although the hypervisor must be involved in data-path aspects (for example, scheduling the CPU between domains, filtering network packets before transmission, or enforcing access control when reading data blocks), there is no need for it to be involved in, or even aware of, higher level issues such as how the CPU is to be shared, or which kinds of packet each domain may transmit.

The resulting architecture is one in which the hypervisor itself provides only basic control operations. These are exported through an interface accessible from authorized domains; potentially complex policy decisions, such as admission control, are best performed by management software running over a guest OS rather than in privileged hypervisor code.

The overall system structure is illustrated in Figure 1. Note that a domain is created at boot time which is permitted to use the *control interface*. This initial domain, termed *Domain0*, is responsible for hosting the application-level management software. The control interface provides the ability to create and terminate other domains and to control their associated scheduling parameters, physical memory allocations and the access they are given to the machine's physical disks and network devices.

In addition to processor and memory resources, the control interface supports the creation and deletion of virtual network interfaces (VIFs) and block devices (VBDs). These virtual I/O devices have associated access-control information which determines which domains can access them, and with what restrictions (for example, a

read-only VBD may be created, or a VIF may filter IP packets to prevent source-address spoofing).

This control interface, together with profiling statistics on the current state of the system, is exported to a suite of application-level management software running in *Domain0*. This complement of administrative tools allows convenient management of the entire server; current tools can create and destroy domains, set network filters and routing rules, monitor per-domain network activity at packet and flow granularity, and create and delete virtual network interfaces and virtual block devices. We anticipate the development of higher-level tools to further automate the application of administrative policy.

3. DETAILED DESIGN

In this section we introduce the design of the major subsystems that make up a Xen-based server. In each case we present both Xen and guest OS functionality for clarity of exposition. The current discussion of guest OSes focuses on XenoLinux as this is the most mature; nonetheless our ongoing porting of Windows XP and NetBSD gives us confidence that Xen is guest OS agnostic.

3.1 Control Transfer: Hypercalls and Events

Two mechanisms exist for control interactions between Xen and an overlying domain: synchronous calls from a domain to Xen may be made using a *hypercall*, while notifications are delivered to domains from Xen using an asynchronous event mechanism.

The hypercall interface allows domains to perform a synchronous software trap into the hypervisor to perform a privileged operation, analogous to the use of system calls in conventional operating systems. An example use of a hypercall is to request a set of page-table updates, in which Xen validates and applies a list of updates, returning control to the calling domain when this is completed.

Communication from Xen to a domain is provided through an asynchronous event mechanism, which replaces the usual delivery mechanisms for device interrupts and allows lightweight notification of important events such as domain-termination requests. Akin to traditional Unix signals, there are only a small number of events, each acting to flag a particular type of occurrence. For instance, events are used to indicate that new data has been received over the network, or that a virtual disk request has completed.

Pending events are stored in a per-domain bitmask which is updated by Xen before invoking an event-callback handler specified by the guest OS. The callback handler is responsible for resetting the set of pending events, and responding to the notifications in an appropriate manner. A domain may explicitly defer event handling by setting a Xen-readable software flag: this is analogous to disabling interrupts on a real processor.

3.2 Data Transfer: I/O Rings

The presence of a hypervisor means there is an additional protection domain between guest OSes and I/O devices, so it is crucial that a data transfer mechanism be provided that allows data to move vertically through the system with as little overhead as possible.

Two main factors have shaped the design of our I/O-transfer mechanism: resource management and event notification. For resource accountability, we attempt to minimize the work required to demultiplex data to a specific domain when an interrupt is received from a device — the overhead of managing buffers is carried out later where computation may be accounted to the appropriate domain. Similarly, memory committed to device I/O is provided by the relevant domains wherever possible to prevent the crosstalk inherent in shared buffer pools; I/O buffers are protected during data transfer by pinning the underlying page frames within Xen.

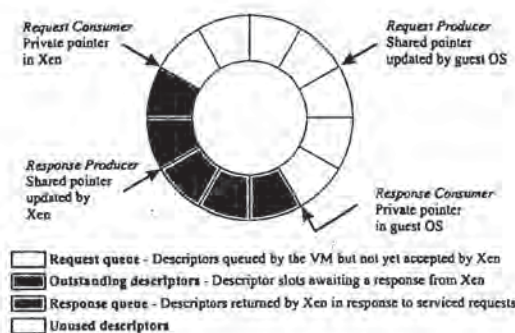


Figure 2: The structure of asynchronous I/O rings, which are used for data transfer between Xen and guest OSes.

Figure 2 shows the structure of our I/O descriptor rings. A ring is a circular queue of descriptors allocated by a domain but accessible from within Xen. Descriptors do not directly contain I/O data; instead, I/O data buffers are allocated out-of-band by the guest OS and indirectly referenced by I/O descriptors. Access to each ring is based around two pairs of producer-consumer pointers: domains place requests on a ring, advancing a request producer pointer, and Xen removes these requests for handling, advancing an associated request consumer pointer. Responses are placed back on the ring similarly, save with Xen as the producer and the guest OS as the consumer. There is no requirement that requests be processed in order: the guest OS associates a unique identifier with each request which is reproduced in the associated response. This allows Xen to unambiguously reorder I/O operations due to scheduling or priority considerations.

This structure is sufficiently generic to support a number of different device paradigms. For example, a set of 'requests' can provide buffers for network packet reception; subsequent 'responses' then signal the arrival of packets into these buffers. Reordering is useful when dealing with disk requests as it allows them to be scheduled within Xen for efficiency, and the use of descriptors with out-of-band buffers makes implementing zero-copy transfer easy.

We decouple the production of requests or responses from the notification of the other party: in the case of requests, a domain may enqueue multiple entries before invoking a hypercall to alert Xen; in the case of responses, a domain can defer delivery of a notification event by specifying a threshold number of responses. This allows each domain to trade-off latency and throughput requirements, similarly to the flow-aware interrupt dispatch in the ArseNIC Gigabit Ethernet interface [34].

3.3 Subsystem Virtualization

The control and data transfer mechanisms described are used in our virtualization of the various subsystems. In the following, we discuss how this virtualization is achieved for CPU, timers, memory, network and disk.

3.3.1 CPU scheduling

Xen currently schedules domains according to the Borrowed Virtual Time (BVT) scheduling algorithm [11]. We chose this particular algorithm since it is both work-conserving and has a special mechanism for low-latency wake-up (or *dispatch*) of a domain when it receives an event. Fast dispatch is particularly important to minimize the effect of virtualization on OS subsystems that are designed to run in a timely fashion; for example, TCP relies on

the timely delivery of acknowledgments to correctly estimate network round-trip times. BVT provides low-latency dispatch by using virtual-time warping, a mechanism which temporarily violates 'ideal' fair sharing to favor recently-woken domains. However, other scheduling algorithms could be trivially implemented over our generic scheduler abstraction. Per-domain scheduling parameters can be adjusted by management software running in *Domain0*.

3.3.2 Time and timers

Xen provides guest OSes with notions of real time, virtual time and wall-clock time. Real time is expressed in nanoseconds passed since machine boot and is maintained to the accuracy of the processor's cycle counter and can be frequency-locked to an external time source (for example, via NTP). A domain's virtual time only advances while it is executing: this is typically used by the guest OS scheduler to ensure correct sharing of its timeslice between application processes. Finally, wall-clock time is specified as an offset to be added to the current real time. This allows the wall-clock time to be adjusted without affecting the forward progress of real time.

Each guest OS can program a pair of alarm timers, one for real time and the other for virtual time. Guest OSes are expected to maintain internal timer queues and use the Xen-provided alarm timers to trigger the earliest timeout. Timeouts are delivered using Xen's event mechanism.

3.3.3 Virtual address translation

As with other subsystems, Xen attempts to virtualize memory access with as little overhead as possible. As discussed in Section 2.1.1, this goal is made somewhat more difficult by the x86 architecture's use of hardware page tables. The approach taken by VMware is to provide each guest OS with a virtual page table, not visible to the memory-management unit (MMU) [10]. The hypervisor is then responsible for trapping accesses to the virtual page table, validating updates, and propagating changes back and forth between it and the MMU-visible 'shadow' page table. This greatly increases the cost of certain guest OS operations, such as creating new virtual address spaces, and requires explicit propagation of hardware updates to 'accessed' and 'dirty' bits.

Although full virtualization forces the use of shadow page tables, to give the illusion of contiguous physical memory, Xen is not so constrained. Indeed, Xen need only be involved in page table updates, to prevent guest OSes from making unacceptable changes. Thus we avoid the overhead and additional complexity associated with the use of shadow page tables — the approach in Xen is to register guest OS page tables directly with the MMU, and restrict guest OSes to read-only access. Page table updates are passed to Xen via a hypercall; to ensure safety, requests are *validated* before being applied.

To aid validation, we associate a type and reference count with each machine page frame. A frame may have any one of the following mutually-exclusive types at any point in time: page directory (PD), page table (PT), local descriptor table (LDT), global descriptor table (GDT), or writable (RW). Note that a guest OS may always create readable mappings to its own page frames, regardless of their current types. A frame may only safely be retasked when its reference count is zero. This mechanism is used to maintain the invariants required for safety; for example, a domain cannot have a writable mapping to any part of a page table as this would require the frame concerned to simultaneously be of types PT and RW.

The type system is also used to track which frames have already been validated for use in page tables. To this end, guest OSes indicate when a frame is allocated for page-table use — this requires a one-off validation of every entry in the frame by Xen, after which

its type is pinned to PD or PT as appropriate, until a subsequent unpin request from the guest OS. This is particularly useful when changing the page table base pointer, as it obviates the need to validate the new page table on every context switch. Note that a frame cannot be retasked until it is both unpinned and its reference count has reduced to zero – this prevents guest OSes from using unpin requests to circumvent the reference-counting mechanism.

To minimize the number of hypercalls required, guest OSes can locally queue updates before applying an entire batch with a single hypercall – this is particularly beneficial when creating new address spaces. However we must ensure that updates are committed early enough to guarantee correctness. Fortunately, a guest OS will typically execute a TLB flush before the first use of a new mapping: this ensures that any cached translation is invalidated. Hence, committing pending updates immediately before a TLB flush usually suffices for correctness. However, some guest OSes elide the flush when it is certain that no stale entry exists in the TLB. In this case it is possible that the first attempted use of the new mapping will cause a page-not-present fault. Hence the guest OS fault handler must check for outstanding updates; if any are found then they are flushed and the faulting instruction is retried.

3.3.4 Physical memory

The initial memory allocation, or *reservation*, for each domain is specified at the time of its creation; memory is thus statically partitioned between domains, providing strong isolation. A maximum-allowable reservation may also be specified: if memory pressure within a domain increases, it may then attempt to claim additional memory pages from Xen, up to this reservation limit. Conversely, if a domain wishes to save resources, perhaps to avoid incurring unnecessary costs, it can reduce its memory reservation by releasing memory pages back to Xen.

XenLinux implements a *balloon driver* [42], which adjusts a domain's memory usage by passing memory pages back and forth between Xen and XenLinux's page allocator. Although we could modify Linux's memory-management routines directly, the balloon driver makes adjustments by using existing OS functions, thus simplifying the Linux porting effort. However, paravirtualization can be used to extend the capabilities of the balloon driver; for example, the out-of-memory handling mechanism in the guest OS can be modified to automatically alleviate memory pressure by requesting more memory from Xen.

Most operating systems assume that memory comprises at most a few large contiguous extents. Because Xen does not guarantee to allocate contiguous regions of memory, guest OSes will typically create for themselves the illusion of contiguous *physical memory*, even though their underlying allocation of *hardware memory* is sparse. Mapping from physical to hardware addresses is entirely the responsibility of the guest OS, which can simply maintain an array indexed by physical page frame number. Xen supports efficient hardware-to-physical mapping by providing a shared translation array that is directly readable by all domains – updates to this array are validated by Xen to ensure that the OS concerned owns the relevant hardware page frames.

Note that even if a guest OS chooses to ignore hardware addresses in most cases, it must use the translation tables when accessing its page tables (which necessarily use hardware addresses). Hardware addresses may also be exposed to limited parts of the OS's memory-management system to optimize memory access. For example, a guest OS might allocate particular hardware pages so as to optimize placement within a physically indexed cache [24], or map naturally aligned contiguous portions of hardware memory using superpages [30].

3.3.5 Network

Xen provides the abstraction of a virtual firewall-router (VFR), where each domain has one or more network interfaces (VIFs) logically attached to the VFR. A VIF looks somewhat like a modern network interface card: there are two I/O rings of buffer descriptors, one for transmit and one for receive. Each direction also has a list of associated rules of the form (*<pattern>*, *<action>*) – if the *pattern* matches then the associated *action* is applied.

Domain0 is responsible for inserting and removing rules. In typical cases, rules will be installed to prevent IP source address spoofing, and to ensure correct demultiplexing based on destination IP address and port. Rules may also be associated with hardware interfaces on the VFR. In particular, we may install rules to perform traditional firewalling functions such as preventing incoming connection attempts on insecure ports.

To transmit a packet, the guest OS simply enqueues a buffer descriptor onto the transmit ring. Xen copies the descriptor and, to ensure safety, then copies the packet header and executes any matching filter rules. The packet payload is not copied since we use scatter-gather DMA; however note that the relevant page frames must be pinned until transmission is complete. To ensure fairness, Xen implements a simple round-robin packet scheduler.

To efficiently implement packet reception, we require the guest OS to exchange an unused page frame for each packet it receives; this avoids the need to copy the packet between Xen and the guest OS, although it requires that page-aligned receive buffers be queued at the network interface. When a packet is received, Xen immediately checks the set of receive rules to determine the destination VIF, and exchanges the packet buffer for a page frame on the relevant receive ring. If no frame is available, the packet is dropped.

3.3.6 Disk

Only *Domain0* has direct unchecked access to physical (IDE and SCSI) disks. All other domains access persistent storage through the abstraction of virtual block devices (VBDs), which are created and configured by management software running within *Domain0*. Allowing *Domain0* to manage the VBDs keeps the mechanisms within Xen very simple and avoids more intricate solutions such as the UDFs used by the Exokernel [23].

A VBD comprises a list of extents with associated ownership and access control information, and is accessed via the I/O ring mechanism. A typical guest OS disk scheduling algorithm will reorder requests prior to enqueueing them on the ring in an attempt to reduce response time, and to apply differentiated service (for example, it may choose to aggressively schedule synchronous metadata requests at the expense of speculative readahead requests). However, because Xen has more complete knowledge of the actual disk layout, we also support reordering within Xen, and so responses may be returned out of order. A VBD thus appears to the guest OS somewhat like a SCSI disk.

A translation table is maintained within the hypervisor for each VBD; the entries within this table are installed and managed by *Domain0* via a privileged control interface. On receiving a disk request, Xen inspects the VBD identifier and offset and produces the corresponding sector address and physical device. Permission checks also take place at this time. Zero-copy data transfer takes place using DMA between the disk and pinned memory pages in the requesting domain.

Xen services *batches* of requests from competing domains in a simple round-robin fashion; these are then passed to a standard elevator scheduler before reaching the disk hardware. Domains may explicitly pass down *reorder barriers* to prevent reordering when this is necessary to maintain higher level semantics (e.g. when us-

ing a write-ahead log). The low-level scheduling gives us good throughput, while the batching of requests provides reasonably fair access. Future work will investigate providing more predictable isolation and differentiated service, perhaps using existing techniques and schedulers [39].

3.4 Building a New Domain

The task of building the initial guest OS structures for a new domain is mostly delegated to *Domain0* which uses its privileged control interfaces (Section 2.3) to access the new domain's memory and inform Xen of initial register state. This approach has a number of advantages compared with building a domain entirely within Xen, including reduced hypervisor complexity and improved robustness (accesses to the privileged interface are sanity checked which allowed us to catch many bugs during initial development).

Most important, however, is the ease with which the building process can be extended and specialized to cope with new guest OSes. For example, the boot-time address space assumed by the Linux kernel is considerably simpler than that expected by Windows XP. It would be possible to specify a fixed initial memory layout for all guest OSes, but this would require additional bootstrap code within every guest OS to lay things out as required by the rest of the OS. Unfortunately this type of code is tricky to implement correctly; for simplicity and robustness it is therefore better to implement it within *Domain0* which can provide much richer diagnostics and debugging support than a bootstrap environment.

4. EVALUATION

In this section we present a thorough performance evaluation of Xen. We begin by benchmarking Xen against a number of alternative virtualization techniques, then compare the total system throughput executing multiple applications concurrently on a single native operating system against running each application in its own virtual machine. We then evaluate the performance isolation Xen provides between guest OSes, and assess the total overhead of running large numbers of operating systems on the same hardware. For these measurements, we have used our XenoLinux port (based on Linux 2.4.21) as this is our most mature guest OS. We expect the relative overheads for our Windows XP and NetBSD ports to be similar but have yet to conduct a full evaluation.

There are a number of preexisting solutions for running multiple copies of Linux on the same machine. VMware offers several commercial products that provide virtual x86 machines on which unmodified copies of Linux may be booted. The most commonly used version is VMware Workstation, which consists of a set of privileged kernel extensions to a 'host' operating system. Both Windows and Linux hosts are supported. VMware also offer an enhanced product called ESX Server which replaces the host OS with a dedicated kernel. By doing so, it gains some performance benefit over the workstation product. ESX Server also supports a paravirtualized interface to the network that can be accessed by installing a special device driver (vmxnet) into the guest OS, where deployment circumstances permit.

We have subjected ESX Server to the benchmark suites described below, but sadly are prevented from reporting quantitative results due to the terms of the product's End User License Agreement. Instead we present results from VMware Workstation 3.2, running on top of a Linux host OS, as it is the most recent VMware product without that benchmark publication restriction. ESX Server takes advantage of its native architecture to equal or outperform VMware Workstation and its hosted architecture. While Xen of course requires guest OSes to be ported, it takes advantage of paravirtualization to noticeably outperform ESX Server.

We also present results for User-mode Linux (UML), an increasingly popular platform for virtual hosting. UML is a port of Linux to run as a user-space process on a Linux host. Like XenoLinux, the changes required are restricted to the architecture dependent code base. However, the UML code bears little similarity to the native x86 port due to the very different nature of the execution environments. Although UML can run on an unmodified Linux host, we present results for the 'Single Kernel Address Space' (skas3) variant that exploits patches to the host OS to improve performance.

We also investigated three other virtualization techniques for running ported versions of Linux on the same x86 machine. Connectix's Virtual PC and forthcoming Virtual Server products (now acquired by Microsoft) are similar in design to VMware's, providing full x86 virtualization. Since all versions of Virtual PC have benchmarking restrictions in their license agreements we did not subject them to closer analysis. UMLinux is similar in concept to UML but is a different code base and has yet to achieve the same level of performance, so we omit the results. Work to improve the performance of UMLinux through host OS modifications is ongoing [25]. Although Plex86 was originally a general purpose x86 VMM, it has now been retargeted to support just Linux guest OSes. The guest OS must be specially compiled to run on Plex86, but the source changes from native x86 are trivial. The performance of Plex86 is currently well below the other techniques.

All the experiments were performed on a Dell 2650 dual processor 2.4GHz Xeon server with 2GB RAM, a Broadcom Tigon 3 Gigabit Ethernet NIC, and a single Hitachi DK32EJ 146GB 10k RPM SCSI disk. Linux version 2.4.21 was used throughout, compiled for architecture *i686* for the native and VMware guest OS experiments, for *x86_64* when running on Xen, and architecture *um* when running on UML. The Xeon processors in the machine support SMT ("hyperthreading"), but this was disabled because none of the kernels currently have SMT-aware schedulers. We ensured that the total amount of memory available to all guest OSes plus their VMM was equal to the total amount available to native Linux.

The RedHat 7.2 distribution was used throughout, installed on ext3 file systems. The VMs were configured to use the same disk partitions in 'persistent raw mode', which yielded the best performance. Using the same file system image also eliminated potential differences in disk seek times and transfer rates.

4.1 Relative Performance

We have performed a battery of experiments in order to evaluate the overhead of the various virtualization techniques relative to running on the 'bare metal'. Complex application-level benchmarks that exercise the whole system have been employed to characterize performance under a range of server-type workloads. Since neither Xen nor any of the VMware products currently support multiprocessor guest OSes (although they are themselves both SMP capable), the test machine was configured with one CPU for these experiments; we examine performance with concurrent guest OSes later. The results presented are the median of seven trials.

The first cluster of bars in Figure 3 represents a relatively easy scenario for the VMMs. The SPEC CPU suite contains a series of long-running computationally-intensive applications intended to measure the performance of a system's processor, memory system, and compiler quality. The suite performs little I/O and has little interaction with the OS. With almost all CPU time spent executing in user-space code, all three VMMs exhibit low overhead.

The next set of bars show the total elapsed time taken to build a default configuration of the Linux 2.4.21 kernel on a local ext3 file system with gcc 2.96. Native Linux spends about 7% of the CPU time in the OS, mainly performing file I/O, scheduling and

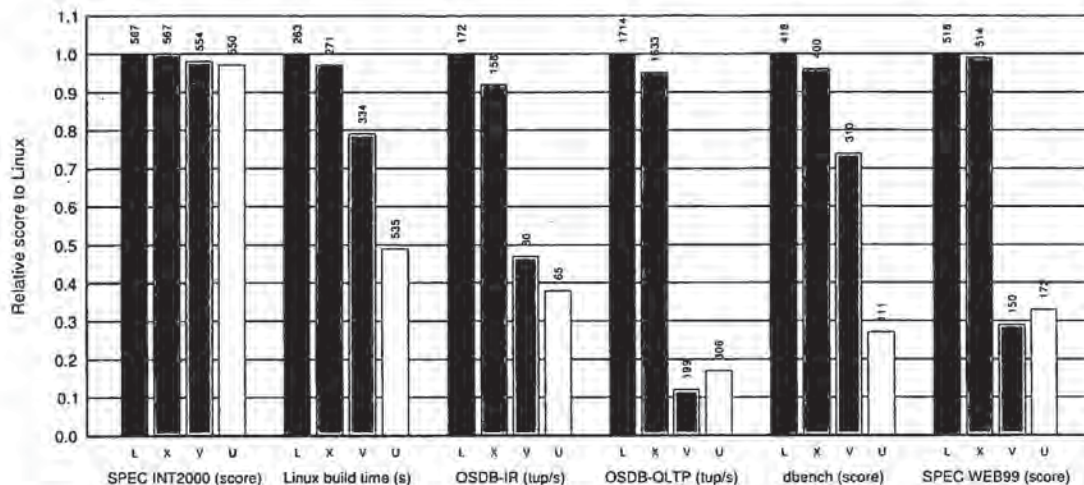


Figure 3: Relative performance of native Linux (L), Xen Linux (X), VMware workstation 3.2 (V) and User-Mode Linux (U).

memory management. In the case of the VMMs, this 'system time' is expanded to a greater or lesser degree: whereas Xen incurs a mere 3% overhead, the other VMMs experience a more significant slowdown.

Two experiments were performed using the PostgreSQL 7.1.3 database, exercised by the Open Source Database Benchmark suite (OSDB) in its default configuration. We present results for the multi-user Information Retrieval (IR) and On-Line Transaction Processing (OLTP) workloads, both measured in tuples per second. A small modification to the suite's test harness was required to produce correct results, due to a UML bug which loses virtual-timer interrupts under high load. The benchmark drives the database via PostgreSQL's native API (callable SQL) over a Unix domain socket, PostgreSQL places considerable load on the operating system, and this is reflected in the substantial virtualization overheads experienced by VMware and UML. In particular, the OLTP benchmark requires many synchronous disk operations, resulting in many protection domain transitions.

The dbench program is a file system benchmark derived from the industry-standard 'NetBench'. It emulates the load placed on a file server by Windows 95 clients. Here, we examine the throughput experienced by a single client performing around 90,000 file system operations.

SPEC WEB99 is a complex application-level benchmark for evaluating web servers and the systems that host them. The workload is a complex mix of page requests: 30% require dynamic content generation, 16% are HTTP POST operations and 0.5% execute a CGI script. As the server runs it generates access and POST logs, so the disk workload is not solely read-only. Measurements therefore reflect general OS performance, including file system and network, in addition to the web server itself.

A number of client machines are used to generate load for the server under test, with each machine simulating a collection of users concurrently accessing the web site. The benchmark is run repeatedly with different numbers of simulated users to determine the maximum number that can be supported. SPEC WEB99 defines a minimum Quality of Service that simulated users must receive in order to be 'conformant' and hence count toward the score: users

must receive an aggregate bandwidth in excess of 320Kb/s over a series of requests. A warm-up phase is allowed in which the number of simultaneous clients is slowly increased, allowing servers to preload their buffer caches.

For our experimental setup we used the Apache HTTP server version 1.3.27, installing the modspecweb99 plug-in to perform most but not all of the dynamic content generation — SPEC rules require 0.5% of requests to use full CGI, forking a separate process. Better absolute performance numbers can be achieved with the assistance of "TUX", the Linux in-kernel static content web server, but we chose not to use this as we felt it was less likely to be representative of our real-world target applications. Furthermore, although Xen's performance improves when using TUX, VMware suffers badly due to the increased proportion of time spent emulating ring 0 while executing the guest OS kernel.

SPEC WEB99 exercises the whole system. During the measurement period there is up to 180Mb/s of TCP network traffic and considerable disk read-write activity on a 2GB dataset. The benchmark is CPU-bound, and a significant proportion of the time is spent within the guest OS kernel, performing network stack processing, file system operations, and scheduling between the many httpd processes that Apache needs to handle the offered load. Xen Linux fares well, achieving within 1% of native Linux performance. VMware and UML both struggle, supporting less than a third of the number of clients of the native Linux system.

4.2 Operating System Benchmarks

To more precisely measure the areas of overhead within Xen and the other VMMs, we performed a number of smaller experiments targeting particular subsystems. We examined the overhead of virtualization as measured by McVoy's *lmbench* program [29]. We used version 3.0-a3 as this addresses many of the issues regarding the fidelity of the tool raised by Seltzer's *hbench* [6]. The OS performance subset of the *lmbench* suite consist of 37 microbenchmarks. In the native Linux case, we present figures for both uniprocessor (L-UP) and SMP (L-SMP) kernels as we were somewhat surprised by the performance overhead incurred by the extra locking in the SMP system in many cases.

Config	null call	null I/O	stat	openslct	closeTCP	sig inst	sig hndl	fork	exec	sh
L-SMP	0.53	0.81	2.10	3.51	23.2	0.83	2.94	143	601	4k2
L-UP	0.45	0.50	1.28	1.92	5.70	0.68	2.49	110	530	4k0
Xen	0.46	0.50	1.22	1.88	5.69	0.69	1.75	198	768	4k8
VMW	0.73	0.83	1.88	2.99	11.1	1.02	4.63	874	2k3	10k
UML	24.7	25.1	36.1	62.8	39.9	26.0	46.0	21k	33k	58k

Table 3: lmbench: Processes - times in μ s

Config	2p 0K	2p 16K	2p 64K	8p 16K	8p 64K	16p 16K	16p 64K
L-SMP	1.69	1.88	2.03	2.36	26.8	4.79	38.4
L-UP	0.77	0.91	1.06	1.03	24.3	3.61	37.6
Xen	1.97	2.22	2.67	3.07	28.7	7.08	39.4
VMW	18.1	17.6	21.3	22.4	51.6	41.7	72.2
UML	15.5	14.6	14.4	16.3	36.8	23.6	52.0

Table 4: lmbench: Context switching times in μ s

Config	0K File create	0K File delete	10K File create	10K File delete	Mmap Prot lat	Page fault
L-SMP	44.9	24.2	123	45.2	99.0	1.33
L-UP	32.1	6.08	66.0	12.5	68.0	1.06
Xen	32.5	5.86	68.2	13.6	139	1.40
VMW	35.3	9.3	85.6	21.4	620	7.53
UML	130	65.7	250	113	1k4	21.8

Table 5: lmbench: File & VM system latencies in μ s

In 24 of the 37 microbenchmarks, XenLinux performs similarly to native Linux, tracking the uniprocessor Linux kernel performance closely and outperforming the SMP kernel. In Tables 3 to 5 we show results which exhibit interesting performance variations among the test systems; particularly large penalties for Xen are shown in bold face.

In the process microbenchmarks (Table 3), Xen exhibits slower *fork*, *exec* and *sh* performance than native Linux. This is expected, since these operations require large numbers of page table updates which must all be verified by Xen. However, the paravirtualization approach allows XenLinux to batch update requests. Creating new page tables presents an ideal case: because there is no reason to commit pending updates sooner, XenLinux can amortize each hypercall across 2048 updates (the maximum size of its batch buffer). Hence each update hypercall constructs 8MB of address space.

Table 4 shows context switch times between different numbers of processes with different working set sizes. Xen incurs an extra overhead between 1 μ s and 3 μ s, as it executes a hypercall to change the page table base. However, context switch results for larger working set sizes (perhaps more representative of real applications) show that the overhead is small compared with cache effects. Unusually, VMware Workstation is inferior to UML on these microbenchmarks; however, this is one area where enhancements in ESX Server are able to reduce the overhead.

The *mmap* latency and *page fault* latency results shown in Table 5 are interesting since they require two transitions into Xen per page: one to take the hardware fault and pass the details to the guest OS, and a second to install the updated page table entry on the guest OS's behalf. Despite this, the overhead is relatively modest.

One small anomaly in Table 3 is that XenLinux has lower signal-handling latency than native Linux. This benchmark does not require any calls into Xen at all, and the 0.75 μ s (30%) speedup is pre-

	TCP MTU 1500		TCP MTU 500	
	TX	RX	TX	RX
Linux	897	897	602	544
Xen	897 (-0%)	897 (-0%)	516 (-14%)	467 (-14%)
VMW	291 (-68%)	615 (-31%)	101 (-83%)	137 (-75%)
UML	165 (-82%)	203 (-77%)	61.1(-90%)	91.4(-83%)

Table 6: *ttcp*: Bandwidth in Mb/s

sumably due to a fortuitous cache alignment in XenLinux, hence underlining the dangers of taking microbenchmarks too seriously.

4.2.1 Network performance

In order to evaluate the overhead of virtualizing the network, we examine TCP performance over a Gigabit Ethernet LAN. In all experiments we use a similarly-configured SMP box running native Linux as one of the endpoints. This enables us to measure receive and transmit performance independently. The *ttcp* benchmark was used to perform these measurements. Both sender and receiver applications were configured with a socket buffer size of 128kB, as we found this gave best performance for all tested systems. The results presented are a median of 9 experiments transferring 400MB.

Table 6 presents two sets of results, one using the default Ethernet MTU of 1500 bytes, the other using a 500-byte MTU (chosen as it is commonly used by dial-up PPP clients). The results demonstrate that the page-flipping technique employed by the XenLinux virtual network driver avoids the overhead of data copying and hence achieves a very low per-byte overhead. With an MTU of 500 bytes, the per-packet overheads dominate. The extra complexity of transmit firewalling and receive demultiplexing adversely impact the throughput, but only by 14%.

VMware emulate a 'pcnet32' network card for communicating with the guest OS which provides a relatively clean DMA-based interface. ESX Server also supports a special 'vmxnet' driver for compatible guest OSes, which provides significant networking performance improvements.

4.3 Concurrent Virtual Machines

In this section, we compare the performance of running multiple applications in their own guest OS against running them on the same native operating system. Our focus is on the results using Xen, but we comment on the performance of the other VMMs where applicable.

Figure 4 shows the results of running 1, 2, 4, 8 and 16 copies of the SPEC WEB99 benchmark in parallel on a two CPU machine. The native Linux was configured for SMP; on it we ran multiple copies of Apache as concurrent processes. In Xen's case, each instance of SPEC WEB99 was run in its own uniprocessor Linux guest OS (along with an *sshd* and other management processes). Different TCP port numbers were used for each web server to enable the copies to be run in parallel. Note that the size of the SPEC data set required for *c* simultaneous connections is $(25 + (c \times 0.66)) \times 4.88$ MBytes or approximately 3.3GB for 1000 connections. This is sufficiently large to thoroughly exercise the disk and buffer cache subsystems.

Achieving good SPEC WEB99 scores requires both high throughput and bounded latency: for example, if a client request gets stalled due to a badly delayed disk read, then the connection will be classed as non conforming and won't contribute to the score. Hence, it is important that the VMM schedules domains in a timely fashion. By default, Xen uses a 5ms time slice.

In the case of a single Apache instance, the addition of a sec-

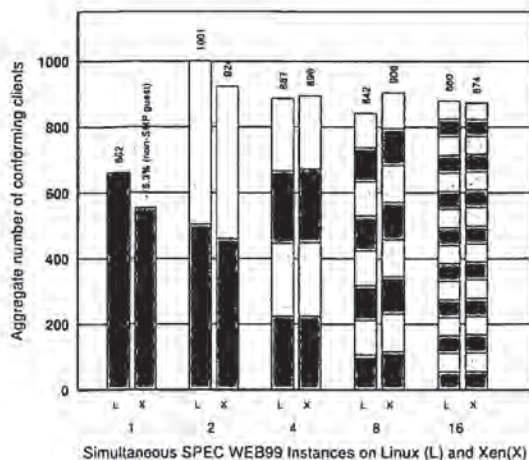


Figure 4: SPEC WEB99 for 1, 2, 4, 8 and 16 concurrent Apache servers: higher values are better.

ond CPU enables native Linux to improve on the score reported in section 4.1 by 28%, to 662 conformant clients. However, the best aggregate throughput is achieved when running two Apache instances, suggesting that Apache 1.3.27 may have some SMP scalability issues.

When running a single domain, Xen is hampered by a lack of support for SMP guest OSes. However, Xen's interrupt load balancer identifies the idle CPU and diverts all interrupt processing to it, improving on the single CPU score by 9%. As the number of domains increases, Xen's performance improves to within a few percent of the native case.

Next we performed a series of experiments running multiple instances of PostgreSQL exercised by the OSDB suite. Running multiple PostgreSQL instances on a single Linux OS proved difficult, as it is typical to run a single PostgreSQL instance supporting multiple databases. However, this would prevent different users having separate database configurations. We resorted to a combination of `chroot` and software patches to avoid SysV IPC name-space clashes between different PostgreSQL instances. In contrast, Xen allows each instance to be started in its own domain allowing easy configuration.

In Figure 5 we show the aggregate throughput Xen achieves when running 1, 2, 4 and 8 instances of OSDB-IR and OSDB-OLTP. When a second domain is added, full utilization of the second CPU almost doubles the total throughput. Increasing the number of domains further causes some reduction in aggregate throughput which can be attributed to increased context switching and disk head movement. Aggregate scores running multiple PostgreSQL instances on a single Linux OS are 25-35% lower than the equivalent scores using Xen. The cause is not fully understood, but it appears that PostgreSQL has SMP scalability issues combined with poor utilization of Linux's block cache.

Figure 5 also demonstrates performance differentiation between 8 domains. Xen's schedulers were configured to give each domain an integer weight between 1 and 8. The resulting throughput scores for each domain are reflected in the different banding on the bar. In the IR benchmark, the weighting has a precise influence over throughput and each segment is within 4% of its expected size.

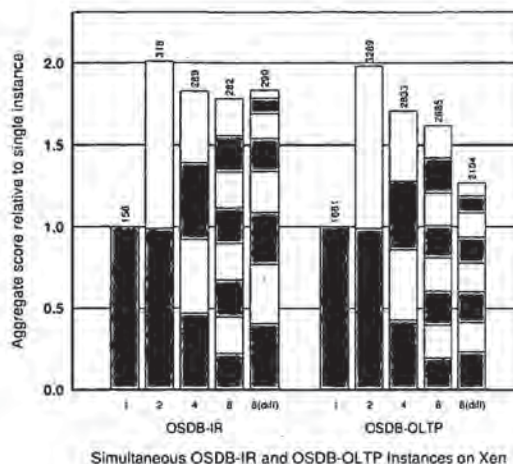


Figure 5: Performance of multiple instances of PostgreSQL running OSDB in separate Xen domains. 8(diff) bars show performance variation with different scheduler weights.

However, in the OLTP case, domains given a larger share of resources to not achieve proportionally higher scores: The high level of synchronous disk activity highlights a weakness in our current disk scheduling algorithm causing them to under-perform.

4.4 Performance Isolation

In order to demonstrate the performance isolation provided by Xen, we hoped to perform a "bakeoff" between Xen and other OS-based implementations of performance isolation techniques such as resource containers. However, at the current time there appear to be no implementations based on Linux 2.4 available for download. QLinux 2.4 has yet to be released and is targeted at providing QoS for multimedia applications rather than providing full defensive isolation in a server environment. Ensism's Linux-based Private Virtual Server product appears to be the most complete implementation, reportedly encompassing control of CPU, disk, network and physical memory resources [14]. We are in discussions with Ensism and hope to be able to report results of a comparative evaluation at a later date.

In the absence of a side-by-side comparison, we present results showing that Xen's performance isolation works as expected, even in the presence of a malicious workload. We ran 4 domains configured with equal resource allocations, with two domains running previously-measured workloads (PostgreSQL/OSDB-IR and SPEC WEB99), and two domains each running a pair of extremely antisocial processes. The third domain concurrently ran a disk bandwidth hog (sustained dd) together with a file system intensive workload targeting huge numbers of small file creations within large directories. The fourth domain ran a 'fork bomb' at the same time as a virtual memory intensive application which attempted to allocate and touch 3GB of virtual memory and, on failure, freed every page and then restarted.

We found that both the OSDB-IR and SPEC WEB99 results were only marginally affected by the behaviour of the two domains running disruptive processes — respectively achieving 4% and 2% below the results reported earlier. We attribute this to the overhead of extra context switches and cache effects. We regard this as

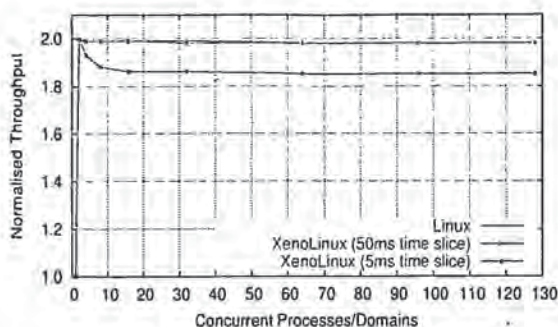


Figure 6: Normalized aggregate performance of a subset of SPEC CINT2000 running concurrently on 1-128 domains

somewhat fortuitous in light of our current relatively simple disk scheduling algorithm, but under this scenario it appeared to provide sufficient isolation from the page-swapping and disk-intensive activities of the other domains for the benchmarks to make good progress. VMware Workstation achieves similar levels of isolation, but at reduced levels of absolute performance.

We repeated the same experiment under native Linux. Unsurprisingly, the disruptive processes rendered the machine completely unusable for the two benchmark processes, causing almost all the CPU time to be spent in the OS.

4.5 Scalability

In this section, we examine Xen's ability to scale to its target of 100 domains. We discuss the memory requirements of running many instances of a guest OS and associated applications, and measure the CPU performance overhead of their execution.

We evaluated the minimum physical memory requirements of a domain booted with XenLinux and running the default set of RH7.2 daemons, along with an `sshd` and Apache web server. The domain was given a reservation of 64MB on boot, limiting the maximum size to which it could grow. The guest OS was instructed to minimize its memory footprint by returning all pages possible to Xen. Without any swap space configured, the domain was able to reduce its memory footprint to 6.2MB; allowing the use of a swap device reduced this further to 4.2MB. A quiescent domain is able to stay in this reduced state until an incoming HTTP request or periodic service causes more memory to be required. In this event, the guest OS will request pages back from Xen, growing its footprint as required up to its configured ceiling.

This demonstrates that memory usage overhead is unlikely to be a problem for running 100 domains on a modern server class machine — far more memory will typically be committed to application data and buffer cache usage than to OS or application text pages. Xen itself maintains only a fixed 20kB of state per domain, unlike other VMMs that must maintain shadow page tables etc.

Finally, we examine the overhead of context switching between large numbers of domains rather than simply between processes. Figure 6 shows the normalized aggregate throughput obtained when running a small subset of the SPEC CINT2000 suite concurrently on between 1 and 128 domains or processes on our dual CPU server. The line representing native Linux is almost flat, indicating that for this benchmark there is no loss of aggregate performance when scheduling between so many processes; Linux identifies them all as compute bound, and schedules them with long time slices of 50ms or more. In contrast, the lower line indicates Xen's throughput

when configured with its default 5ms maximum scheduling 'slice'. Although operating 128 simultaneously compute bound processes on a single server is unlikely to be commonplace in our target application area, Xen copes relatively well: running 128 domains we lose just 7.5% of total throughput relative to Linux.

Under this extreme load, we measured user-to-user UDP latency to one of the domains running the SPEC CINT2000 subset. We measured mean response times of 147ms (standard deviation 97ms). Repeating the experiment against a 129th domain that was otherwise idle, we recorded a mean response time of 5.4ms (s.d. 16ms). These figures are very encouraging — despite the substantial background load, interactive domains remain responsive.

To determine the cause of the 7.5% performance reduction, we set Xen's scheduling 'slice' to 50ms (the default value used by ESX Server). The result was a throughput curve that tracked native Linux's closely, almost eliminating the performance gap. However, as might be expected, interactive performance at high load is adversely impacted by these settings.

5. RELATED WORK

Virtualization has been applied to operating systems both commercially and in research for nearly thirty years. IBM VM/370 [19, 38] first made use of virtualization to allow binary support for legacy code. VMware [10] and Connectix [8] both virtualize commodity PC hardware, allowing multiple operating systems to run on a single host. All of these examples implement a full virtualization of (at least a subset of) the underlying hardware, rather than paravirtualizing and presenting a modified interface to the guest OS. As shown in our evaluation, the decision to present a full virtualization, although able to more easily support off-the-shelf operating systems, has detrimental consequences for performance.

The virtual machine monitor approach has also been used by Disco to allow commodity operating systems to run efficiently on ccNUMA machines [7, 18]. A small number of changes had to be made to the hosted operating systems to enable virtualized execution on the MIPS architecture. In addition, certain other changes were made for performance reasons.

At present, we are aware of two other systems which take the paravirtualization approach: IBM presently supports a paravirtualized version of Linux for their zSeries mainframes, allowing large numbers of Linux instances to run simultaneously. Denali [44], discussed previously, is a contemporary isolation kernel which attempts to provide a system capable of hosting vast numbers of virtualized OS instances.

In addition to Denali, we are aware of two other efforts to use low-level virtualization to build an infrastructure for distributed systems. The vMatrix [1] project is based on VMware and aims to build a platform for moving code between different machines. As vMatrix is developed above VMware, they are more concerned with higher-level issues of distribution than those of virtualization itself. In addition, IBM provides a "Managed Hosting" service, in which virtual Linux instances may be rented on IBM mainframes.

The PlanetLab [33] project has constructed a distributed infrastructure which is intended to serve as a testbed for the research and development of geographically distributed network services. The platform is targeted at researchers and attempts to divide individual physical hosts into *slivers*, providing simultaneous low-level access to users. The current deployment uses VServers [17] and SILK [4] to manage sharing within the operating system.

We share some motivation with the operating system extensibility and active networks communities. However, when running over Xen there is no need to check for "safe" code, or for guaranteed termination — the only person hurt in either case is the client

in question. Consequently, Xen provides a more general solution: there is no need for hosted code to be digitally signed by a trusted compiler (as in SPIN [5]), to be accompanied by a safety proof (as with PCC [31]), to be written in a particular language (as in SafetyNet [22] or any Java-based system), or to rely on a particular middleware (as with mobile-agent systems). These other techniques can, of course, continue to be used within guest OSes running over Xen. This may be particularly useful for workloads with more transient tasks which would not provide an opportunity to amortize the cost of starting a new domain.

A similar argument can be made with regard to language-level virtual machine approaches: while a resource-managed JVM [9] should certainly be able to host untrusted applications, these applications must necessarily be compiled to Java bytecode and follow that particular system's security model. Meanwhile, Xen can readily support language-level virtual machines as applications running over a guest OS.

6. DISCUSSION AND CONCLUSION

We have presented the Xen hypervisor which partitions the resources of a computer between domains running guest operating systems. Our paravirtualizing design places a particular emphasis on performance and resource management. We have also described and evaluated XenLinux, a fully-featured port of a Linux 2.4 kernel that runs over Xen.

6.1 Future Work

We believe that Xen and XenLinux are sufficiently complete to be useful to a wider audience, and so intend to make a public release of our software in the very near future. A beta version is already under evaluation by selected parties; once this phase is complete, a general 1.0 release will be announced on our project page³.

After the initial release we plan a number of extensions and improvements to Xen. To increase the efficiency of virtual block devices, we intend to implement a shared *universal buffer cache* indexed on block contents. This will add controlled data sharing to our design without sacrificing isolation. Adding copy-on-write semantics to virtual block devices will allow them to be safely shared among domains, while still allowing divergent file systems.

To provide better physical memory performance, we plan to implement a *last-chance page cache* (LPC) — effectively a system-wide list of free pages, of non-zero length only when machine memory is undersubscribed. The LPC is used when the guest OS virtual memory system chooses to evict a clean page; rather than discarding this completely, it may be added to the tail of the free list. A fault occurring for that page before it has been reallocated by Xen can therefore be satisfied without a disk access.

An important role for Xen is as the basis of the *XenoServer* project which looks beyond individual machines and is building the control systems necessary to support an Internet-scale computing infrastructure. Key to our design is the idea that resource usage be accounted precisely and paid for by the sponsor of that job — if payments are made in real cash, we can use a congestion pricing strategy [28] to handle excess demand, and use excess revenues to pay for additional machines. This necessitates accurate and timely I/O scheduling with greater resilience to hostile workloads. We also plan to incorporate accounting into our block storage architecture by creating leases for virtual block devices.

In order to provide better support for the management and administration of XenoServers, we are incorporating more thorough support for auditing and forensic logging. We are also developing

additional VFR rules which we hope will allow us to detect and prevent a wide range of antisocial network behaviour. Finally, we are continuing our work on XenoXP, focusing in the first instance on writing network and block device drivers, with the aim of fully supporting enterprise servers such as IIS.

6.2 Conclusion

Xen provides an excellent platform for deploying a wide variety of network-centric services, such as local mirroring of dynamic web content, media stream transcoding and distribution, multiplayer game and virtual reality servers, and 'smart proxies' [2] to provide a less ephemeral network presence for transiently-connected devices.

Xen directly addresses the single largest barrier to the deployment of such services: the present inability to host transient servers for short periods of time and with low instantiation costs. By allowing 100 operating systems to run on a single server, we reduce the associated costs by two orders of magnitude. Furthermore, by turning the setup and configuration of each OS into a software concern, we facilitate much smaller-granularity timescales of hosting.

As our experimental results show in Section 4, the performance of XenLinux over Xen is practically equivalent to the performance of the baseline Linux system. This fact, which comes from the careful design of the interface between the two components, means that there is no appreciable cost in having the resource management facilities available. Our ongoing work to port the BSD and Windows XP kernels to operate over Xen is confirming the generality of the interface that Xen exposes.

Acknowledgments

This work is supported by ESPRC Grant GR/S01894/01 and by Microsoft. We would like to thank Evangelos Kotsovinos, Anil Madhavapeddy, Russ Ross and James Scott for their contributions.

7. REFERENCES

- [1] A. Awadallah and M. Rosenblum. The vMatrix: A network of virtual machine monitors for dynamic content distribution. In *Proceedings of the 7th International Workshop on Web Content Caching and Distribution (WCW 2002)*, Aug. 2002.
- [2] A. Bakre and B. R. Badrinath. I-TCP: indirect TCP for mobile hosts. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS 1995)*, pages 136–143, June 1995.
- [3] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI 1999)*, pages 45–58, Feb. 1999.
- [4] A. Bavier, T. Voigt, M. Wawrzoniak, L. Peterson, and P. Gunningberg. SILK: Scout paths in the Linux kernel. Technical Report 2002-009, Uppsala University, Department of Information Technology, Feb. 2002.
- [5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Flaczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles*, volume 29(5) of *ACM Operating Systems Review*, pages 267–284, Dec. 1995.
- [6] A. Brown and M. Seltzer. Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 1997.
- [7] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles*, volume 31(5) of *ACM Operating Systems Review*, pages 143–156, Oct. 1997.

³<http://www.cl.cam.ac.uk/netos/xen>

- [8] Connectix. Product Overview: Connectix Virtual Server, 2003. <http://www.connectix.com/products/vs.html>.
- [9] G. Czajkowski and L. Dayné. Multitasking without compromise: a virtual machine evolution. *ACM SIGPLAN Notices*, 36(11):125–138, Nov. 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2001).
- [10] S. Devine, E. Bugnion, and M. Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. *US Patent*, 6397242, Oct. 1998.
- [11] K. J. Duda and D. R. Cheriton. Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM SIGOPS Symposium on Operating Systems Principles*, volume 33(5) of *ACM Operating Systems Review*, pages 261–276, Kiawah Island Resort, SC, USA, Dec. 1999.
- [12] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, *ACM Operating Systems Review*, Winter 2002 Special Issue, pages 211–224, Boston, MA, USA, Dec. 2002.
- [13] D. Engler, S. K. Gupta, and F. Kaashoek. AVM: Application-level virtual memory. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 72–77, May 1995.
- [14] Ensim. Ensim Virtual Private Servers, 2003. http://www.ensim.com/products/materials/datasheet_vps_051003.pdf.
- [15] K. A. Fraser, S. M. Hand, T. L. Harris, I. M. Leslie, and I. A. Pratt. The Xenoserver computing infrastructure. Technical Report UCAM-CL-TR-552, University of Cambridge, Computer Laboratory, Jan. 2003.
- [16] T. Garfinkel, M. Rosenblum, and D. Boneh. Flexible OS Support and Applications for Trusted Computing. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, Kauai, Hawaii, May 2003.
- [17] J. Gelineas. Virtual Private Servers and Security Contexts, 2003. http://www.solucorp.qc.ca/misc/prj/s_context.html.
- [18] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 17th ACM SIGOPS Symposium on Operating Systems Principles*, volume 33(5) of *ACM Operating Systems Review*, pages 154–169, Dec. 1999.
- [19] P. H. Gum. System/370 extended architecture: facilities for virtual machines. *IBM Journal of Research and Development*, 27(6):530–544, Nov. 1983.
- [20] S. Hand. Self-paging in the Nemesis operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI 1999)*, pages 73–86, Oct. 1999.
- [21] S. Hand, T. L. Harris, E. Kotsovinos, and I. Pratt. Controlling the XenoServer Open Platform, April 2003.
- [22] A. Jeffrey and I. Wakeman. A Survey of Semantic Techniques for Active Networks, Nov. 1997. <http://www.cogs.susx.ac.uk/projects/safetynet/>.
- [23] M. F. Kaashoek, D. R. Engler, G. R. Granger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles*, volume 31(5) of *ACM Operating Systems Review*, pages 52–65, Oct. 1997.
- [24] R. Kessler and M. Hill. Page placement algorithms for large real-indexed caches. *ACM Transaction on Computer Systems*, 10(4):338–359, Nov. 1992.
- [25] S. T. King, G. W. Dunlap, and P. M. Chen. Operating System Support for Virtual Machines. In *Proceedings of the 2003 Annual USENIX Technical Conference*, Jun 2003.
- [26] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications*, Calicoon, NY, Jun 2002.
- [27] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas In Communications*, 14(7):1280–1297, Sept. 1996.
- [28] J. MacKie-Mason and H. Varian. Pricing congestible network resources. *IEEE Journal on Selected Areas In Communications*, 13(7):1141–1149, Sept. 1995.
- [29] L. McVoy and C. Staelin. Imbench: Portable tools for performance analysis. In *Proceedings of the USENIX Annual Technical Conference*, pages 279–294, Berkeley, Jan. 1996. Usenix Association.
- [30] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, *ACM Operating Systems Review*, Winter 2002 Special Issue, pages 89–104, Boston, MA, USA, Dec. 2002.
- [31] G. C. Necula. Proof-carrying code. In *Conference Record of POPL 1997: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Jan. 1997.
- [32] S. Oikawa and R. Rajkumar. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of the IEEE Real Time Technology and Applications Symposium*, pages 111–120, June 1999.
- [33] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proceedings of the 1st Workshop on Hot Topics in Networks (HotNets-I)*, Princeton, NJ, USA, Oct. 2002.
- [34] I. Pratt and K. Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM-01)*, pages 67–76, Los Alamitos, CA, USA, Apr. 22–26 2001. IEEE Computer Society.
- [35] D. Reed, I. Pratt, P. Menage, S. Early, and N. Stratford. Xenoservers: accounted execution of untrusted code. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, 1999.
- [36] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium’s ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, USA, pages 129–144, Aug. 2000.
- [37] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, *ACM Operating Systems Review*, Winter 2002 Special Issue, pages 377–390, Boston, MA, USA, Dec. 2002.
- [38] L. Seawright and R. MacKinnon. VM/370 – a study of multiplicity and usefulness. *IBM Systems Journal*, pages 4–17, 1979.
- [39] P. Shenoy and H. Vin. Cello: A Disk Scheduling Framework for Next-generation Operating Systems. In *Proceedings of ACM SIGMETRICS’98, the International Conference on Measurement and Modeling of Computer Systems*, pages 44–55, June 1998.
- [40] V. Sundaram, A. Chandra, P. Goyal, P. Shenoy, J. Sahni, and H.M. Vin. Application Performance in the QLinux Multimedia Operating System. In *Proceedings of the 8th ACM Conference on Multimedia*, Nov. 2000.
- [41] D. Tennenhouse. Layered Multiplexing Considered Harmful. In Rudin and Williamson, editors, *Protocols for High-Speed Networks*, pages 143–148. North Holland, 1989.
- [42] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, *ACM Operating Systems Review*, Winter 2002 Special Issue, pages 181–194, Boston, MA, USA, Dec. 2002.
- [43] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Technical Report 02-02-01, University of Washington, 2002.
- [44] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, *ACM Operating Systems Review*, Winter 2002 Special Issue, pages 195–210, Boston, MA, USA, Dec. 2002.

Linux Traffic Control

***Author: Ivan Delchev
Course: Networks and Distributed Systems Seminar
Instructor: Prof. Juergen Schoenwaelder
International University Bremen, Spring 2006***

Processing of Network Data

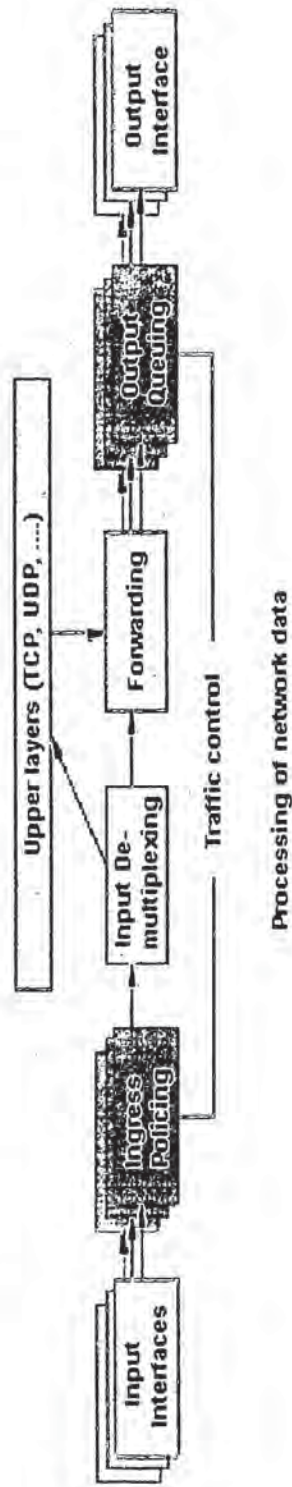


Figure 2.1.4

Fig. 1: Processing of network data by the linux kernel [2]

- Packets arrive via an input interface
- **Input Traffic Control** (policing)
- Input De-Multiplexing – determine whether packet is for local node
- "Forwarding" selects the output interface, the next hop etc.
- **Output Traffic Control**
- Device driver picks up packets and emits them on the network.

Definition of Traffic Control(TC)

- Sets of queuing systems and mechanisms by which packets are received and transmitted on a router
 - Determine which packets to accept and at what rate on the input of an interface
 - Determine which packets to transmit in what order and at what rate on the output of an interface

Why Use Traffic Control?

- Lack of differentiation between types of flows.
- Traffic control allows an administrator to queue packets differently based on attributes of the packets leading to:
- More predictable usage of network resources
- Less volatile contention for network resources

Common Traffic Control Solutions

- Limit total bandwidth to a known rate
- Limit the bandwidth of a particular user
- Reserve bandwidth for a particular application
- Prefer latency sensitive traffic
- Ensure that a particular type of traffic is dropped

Terms

- Queue – a buffer containing a finite number of items waiting for an action or service. Typically FIFO.
- Flow - a distinct connection or conversation between two hosts. Any unique set of packets between two hosts can be regarded as a flow.
- Token - A particular message or bit pattern that signifies permission to transmit.
- Bucket - type of data buffer, containing a number of tokens

Traditional Elements of TC

- Shaping - delay packets to meet a desired rate.
- **Scheduling** - arrange and/or rearrange packets for output
- Classifying - sort or separate traffic into queues for different treatment
- Policing - measure and limit traffic in a particular queue
- Dropping - discard an entire packet, flow or classification
- Marking - a mechanism by which a packet is altered.

Classless Queuing Disciplines 1

- Classless qdisc - a qdisc with no configurable internal subdivisions
- FIFO (pfifo and bfifo)
- FIFO Fast(pfifo_fast)

pfifo_fast queuing discipline

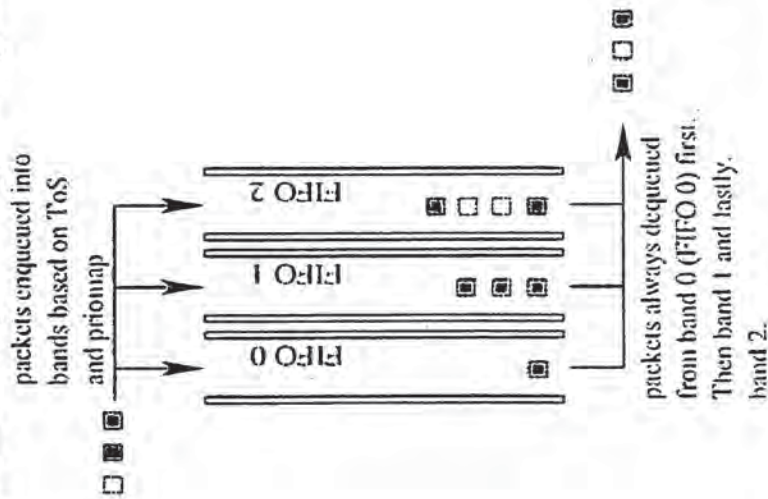


Fig. 2 FIFO Fast queuing discipline[1]

Classless Queuing Disciplines 2

Stochastic Fair Queuing (SFQ)

- Stochastic Fair Queuing(SFQ)
- Goal - fairly distribute opportunity to transmit data among flows
- Uses a hash function to separate the traffic
- Dequeue queues in a round-robin fashion
- Perturb hash function periodically to avoid unfairness

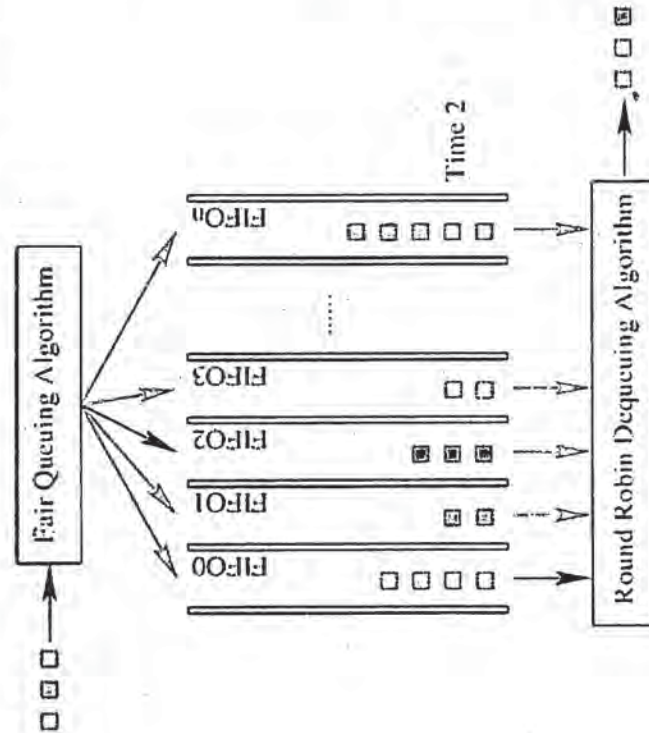


Fig. 3 Stochastic Fair Queuing [1]

Classless Queuing Disciplines 3

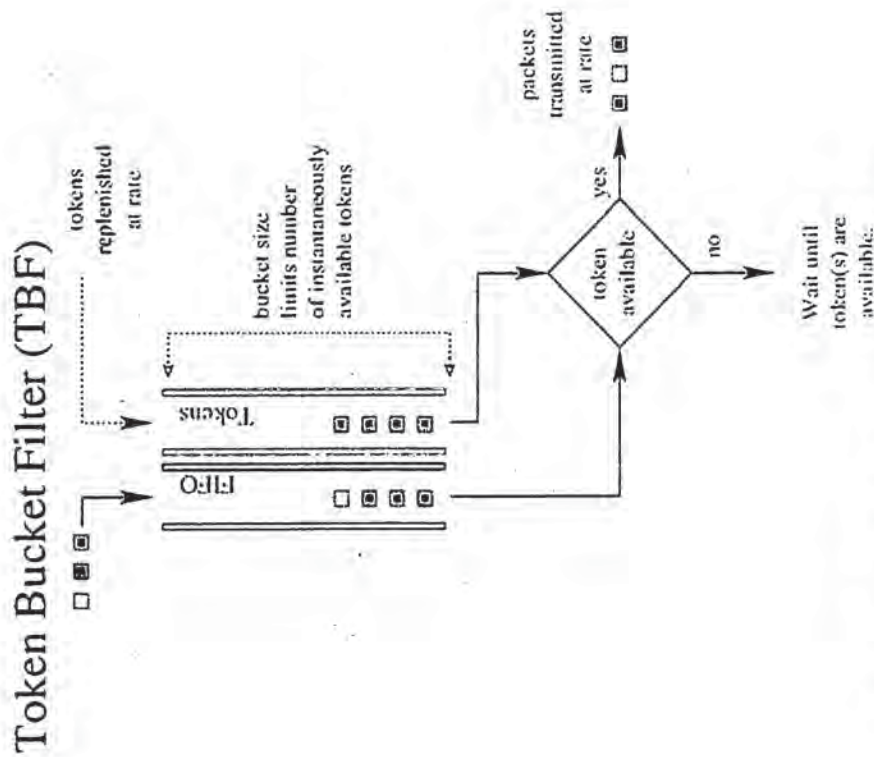


Fig. 4 Token Bucket Filter [1]

- Token Bucket Filter(TBF)
- Goal - shape/limit traffic transmitted on an interface
- Generate tokens at a desired rate
- Send packets only if there are sufficient tokens available.

Classful Queuing Disciplines 1

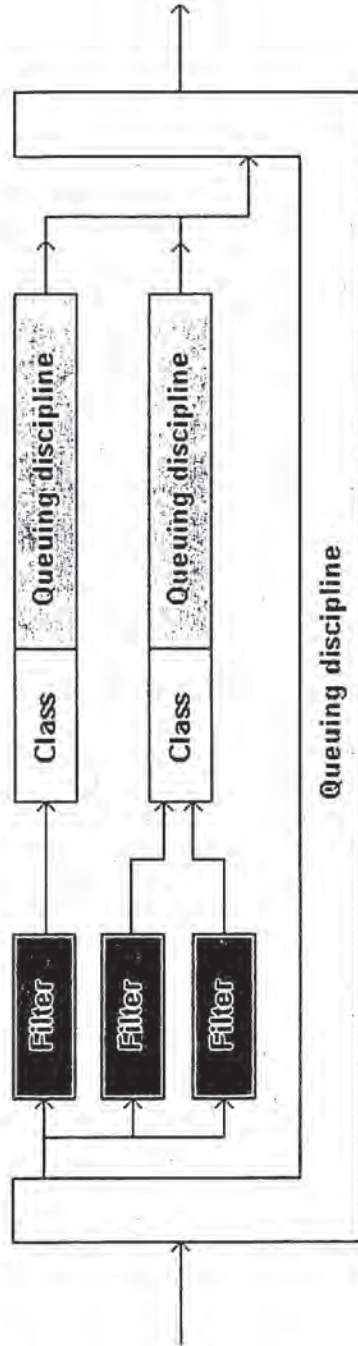


Figure 2.1.7

Fig. 5: Example of a queuing discipline containing filters and classes [2]

- Classful qdiscs can contain filters and classes
- Filters distinguish among different classes of packets and process each class in a specific way – e.g. pass to different internal class object
- The queuing discipline on the respective class takes

Classful Queuing Disciplines 2

- Hierarchical Token Bucket (HTB)
 - HTB is an arbitrary number of token buckets arranged in a hierarchy
 - User can define the characteristics of the tokens and buckets used and to nest these buckets in an arbitrary fashion.
 - “rate” - sets the guaranteed bandwidth available for a given class (“tokens” bucket)
 - “ceil” - sets the maximum bandwidth that class is allowed to consume. (“ctokens” bucket)

Classful Queuing Disciplines 3 – HTB Contd.

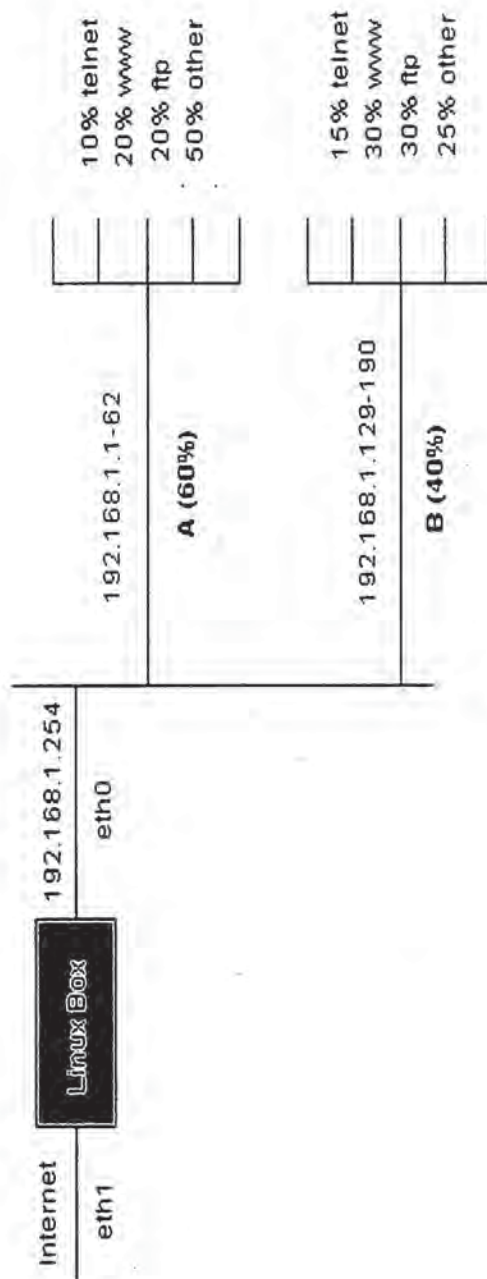
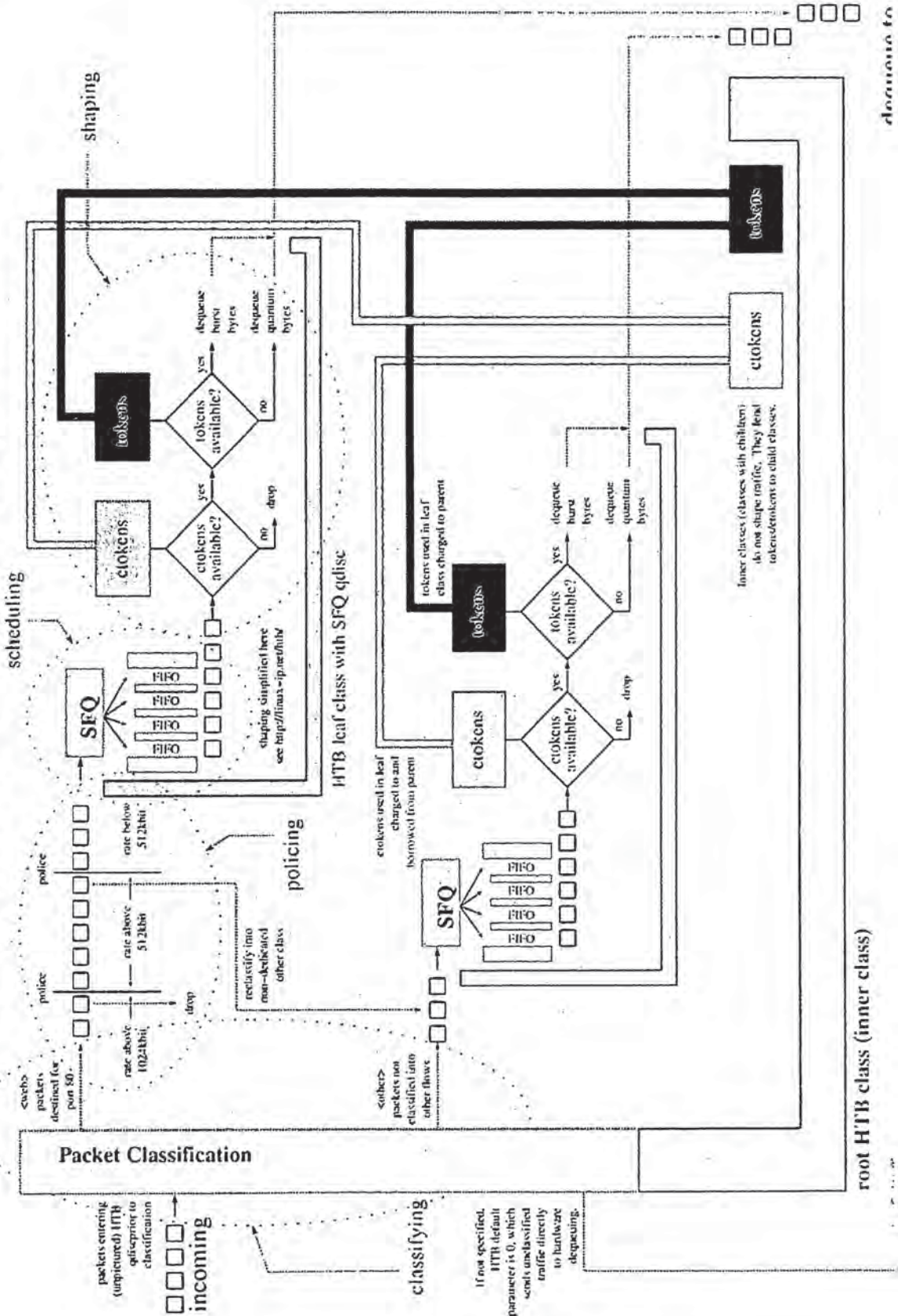


Figure 2.8.3

Fig. 6 Hierarchical link-sharing structure using HTB [2]

- Use one physical link to simulate several slower links and send different kinds of traffic on different simulated links

Simplified Linux Traffic Control Scenario with HTB



References

- [1] <http://www.roseindia.net/linux/tutorial/linux-howto/Traffic-Control-HOWTO/index.html>
- [2] <http://www.opalsoft.net/qos/>



Published on **O'Reilly Network** (<http://www.oreillynet.com/>)
<http://www.oreillynet.com/pub/a/network/2002/02/04/windows.html>
See [this](#) if you're having trouble printing code examples



An Introduction to the InfiniBand Architecture

by Odysseas Pentakalos, coauthor of Windows 2000 Performance Guide
02/04/2002

In 1965, Dr. Gordon Moore observed that the industry was able to double the transistor density on a manufactured die every year (Gordon E. Moore, "The Continuing Silicon Technology Evolution Inside the PC Platform"). This observation became popular as Moore's Law and almost 40 years later it still holds as a fairly accurate estimate of the growth of transistor density (a more accurate estimate of the growth rate, which encompasses growth data from the past 45 years, is doubling of the density every 18 months).

This fast-paced growth in transistor density translates into CPU-performance increases of a similar magnitude, making applications such as data mining, data warehousing, and e-business commonplace. To reap the benefits of this growth in computational power, however, requires that the I/O subsystem is able to deliver the data needed by the processor subsystem at the rate at which it is needed. In the past couple of years, it has become clear that the current shared bus-based architecture will become the bottleneck of the servers that host these powerful but demanding applications.

Performance is just one dimension of the growing demands imposed on the I/O subsystem. As Wendy Vittori, general manager of Intel's I/O products division, said, "The growth of e-Commerce and e-Business means more data delivered to more and more users. This data needs to move faster, with higher reliability and quality of service than ever before." (Intel, I/O Building Blocks, Ultimate I/O Performance.) E-commerce and e-business applications need to be available 24/7 to process transactions at very high rates. This implies that the desired solution to the architecture of the I/O system must be able to provide enhanced reliability and availability in addition to raw performance.

In this article, I will review each of the main architectural features of the InfiniBand as a solution to the corresponding limitation of the current I/O subsystem. The InfiniBand specification defines the architecture of the interconnect that will pull together the I/O subsystems of the next generation of servers and will ultimately even move to the powerful desktops of the future. The architecture is based on a serial, switched fabric that in addition to defining link bandwidths between 2.5 and 30 Gbits/sec, resolves the scalability, expandability, and fault tolerance limitations of the shared bus architecture through the use of switches and routers in the construction of its fabric. But before we delve into that discussion let's see who is behind this new I/O architecture and how it came about.

Who is working on it?

A few years ago, when the impending bottleneck at the I/O subsystem became a clear vision of the

future, a number of industry leaders decided to take action and come up with the design for the I/O subsystem of the future. As it is common in the computer industry, two competing efforts started, one called the Future I/O initiative, which included Compaq, IBM, and Hewlett Packard; and the other one called the Next-Generation I/O initiative, which included Dell, Hitachi, Intel, NEC, Siemens, and Sun Microsystems. As the first versions of those specifications started to become available, the two groups decided to unify their efforts by bringing together the best ideas of each of the two separate initiatives. So, in August of 1999, the seven industry leaders, Compaq, Dell, Hewlett-Packard, IBM, Intel, Microsoft, and Sun Microsystems formed the InfiniBand Trade Association (ITA) (InfiniBand Trade Association, "[What is the InfiniBand Trade Association.](#)").

In addition to those seven steering committee members, the ITA consists of 11 sponsoring members, including among others 3Com, Cisco and EMC, and more than 200 member companies. By agreeing to join their efforts, the ITA was both able to eliminate the confusion in the market that would have resulted from the coexistence of the two competing standards but it was also able to release a first version of the specification very quickly. Version 1.0 of the InfiniBand Architecture Specification was released in October of 2000 and the 1.0a version, mainly consisting of minor changes to the 1.0 version, was released in June of 2001. The specification is available for download from the ITA Web site for free, even for non-members.

What are its main architectural features?

The Peripheral Component Interconnect (PCI) bus, which was first introduced in the early 90's, is the dominant bus, used in both desktop and server machines for attaching I/O peripherals to the CPU/memory complex. The most common configuration of the PCI bus is a 32-bit 33MHz version that provides a bandwidth of 133MB per second, although the 2.2 version of the specification allows for a 64-bit version at 33MHz for a bandwidth of 266MB per second and even a 64-bit 66MHz version for a bandwidth of 533MB per second. Even today's powerful desktop machines have lots of capacity available with the PCI bus in the typical configuration, but server machines are starting to hit the upper limits of the shared bus architecture. The availability of multiport Gigabit Ethernet NICs, along with one or more Fibre Channel I/O controllers can easily consume even the highest 64-bit, 66MHz version of the PCI bus.

To resolve this limitation on the bandwidth of the PCI bus, a number of solutions are becoming available in the market as interim solutions such as PCI-X and PCI DDR (Mellanox Technologies, "[Understanding PCI Bus, 3GIO, and InfiniBand Architecture](#)"). Both of them are backwards compatible upgrade paths to the current PCI bus. The PCI-X specification allows for a 64-bit version of the bus operating at the clock rate of 133 MHz, but this is achieved by easing some of the timing constraints. The shared bus nature of the PCI-X bus forces it to lower its fanout in order to achieve the high clock rate of 133 MHz.

A PCI-X system that is running at 133 MHz can have only one slot on the bus, two PCI-X slots would allow a maximum clock rate of 100 MHz, whereas the four slot configuration would drop down to a clock rate of 6 MHz (Compaq Computer Corporation, "[PCI-X: An Evolution of the PCI Bus](#)," September 1999, TC990903TB.). So, despite the temporary resolution of the PCI bandwidth limitation through these new upgrade technologies, there is a long term solution needed that cannot rely on a shared bus architecture.

InfiniBand breaks through the bandwidth and fanout limitations of the PCI bus by migrating from the traditional shared bus architecture into a switched fabric architecture. Figure 1, below, shows the simplest configuration of an InfiniBand installation, where two or more nodes are connected to one

another through the fabric. A node represents either a host device such as a server or an I/O device such as a RAID subsystem. The fabric itself may consist of a single switch in the simplest case or a collection of interconnected switches and routers. I will describe the difference between a switch and a router a little later but those of you with a networking background have probably guessed already what the difference is.

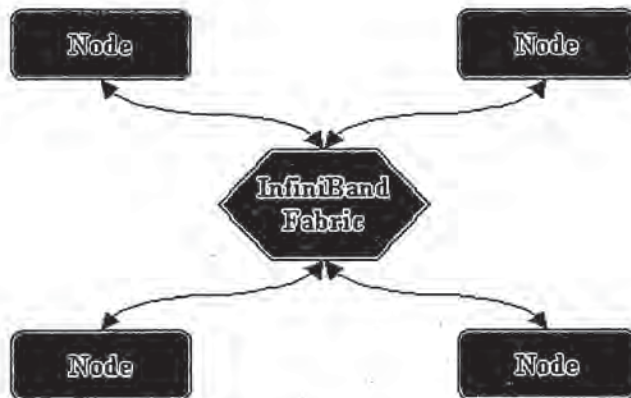


Figure 1. InfiniBand Basic Fabric Topology

Each connection between nodes, switches, and routers is a point-to-point, serial connection. This basic difference brings about a number of benefits:

- Because it is a serial connection, it only requires four as opposed to the wide parallel connection of the PCI bus.
- The point-to-point nature of the connection provides the full capacity of the connection to the two endpoints because the link is dedicated to the two endpoints. This eliminates the contention for the bus as well as the resulting delays that emerge under heavy loading conditions in the shared bus architecture.
- The InfiniBand channel is designed for connections between hosts and I/O devices within a Data Center. Due to the well defined, relatively short length of the connections, much higher bandwidth can be achieved than in cases where much longer lengths may be needed.

The InfiniBand specification defines the raw bandwidth of the base 1x connection at 2.5Gb per second. It then specifies two additional bandwidths, referred to as 4x and 12x, as multipliers of the base link rate. At the time that I am writing this, there are already 1x and 4x adapters available in the market. So, the InfiniBand will be able to achieve much higher data transfer rates than is physically possible with the shared bus architecture without the fan-out limitations of the later.

Lets now dig in a little deeper into the architecture of the InfiniBand to explore some of its additional benefits. Figure 2, below, illustrates a system area network that utilizes the InfiniBand architecture. In the example shown in the figure, the fabric consists of three switches that connect the six end nodes together. Each node connects to the fabric through a channel adapter. The InfiniBand specification classifies the channel adapters into two categories: Host Channel Adapters (HCA) and Target Channel Adapters (TCA).

HCAs are present in servers or even desktop machines and provide an interface that is used to integrate

the InfiniBand with the operating system. TCAs are present on I/O devices such as a RAID subsystem or a JBOD subsystem. Each channel adapter may have one or more ports. As you can also see in the figure, a channel adapter with more than one port, may be connected to multiple switch ports. This allows for multiple paths between a source and a destination, resulting in performance and reliability benefits.

By having multiple paths available in getting the data from one node to another, the fabric is able to achieve transfer rates at the full capacity of the channel, avoiding congestion issues that arise in the shared bus architecture. Furthermore, having alternative paths results in increased reliability and availability since another path is available for routing of the data in the case of failure of one of the links.

Two more unique features of the InfiniBand Architecture that become evident in Figure 2 are the ability to share storage devices across multiple servers and the ability to perform third-party I/O. Third-party I/O is a term used to refer to the ability of two storage devices to complete an I/O transaction without the direct involvement of a host other than in setting up the operation. This feature is extremely important from the performance perspective since many such I/O operations between two storage devices can be totally off-loaded from the server, thereby eliminating the unnecessary CPU utilization that would otherwise be consumed.

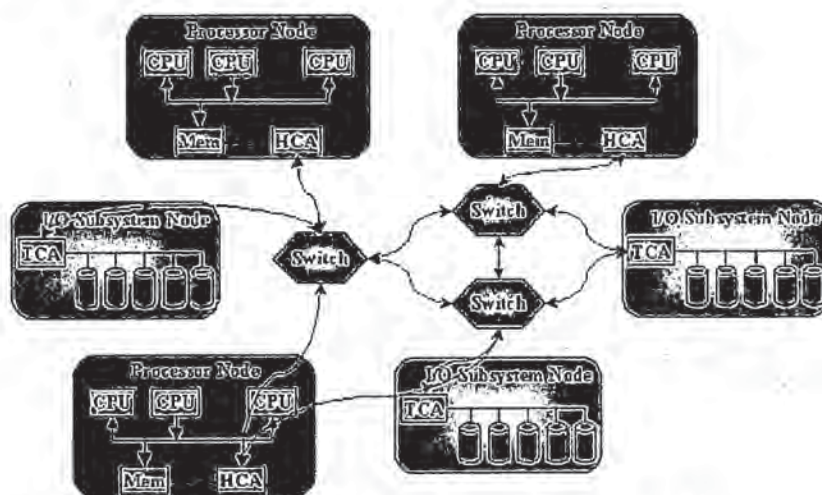


Figure 2. System Area Network based on the InfiniBand Architecture

Host and Target Channel adapters present an interface to the layers above them that allow those layers to generate and consume packets. In the case of a server writing a file to a storage device, the host is generating the packets that are then consumed by the storage device. In contrast to the channel adapter, switches simply forward packets between two of their ports based on the established routing table and the addressing information stored on the packets. A collection of end nodes connected to one another through one or more switches form a subnet. Each subnet must have at least one Subnet Manager that is responsible for the configuration and management of the subnet.

Routers are like switches in the respect that they simply forward packets between their ports. The difference between the routers and the switches, however, is that a router is used

Related Reading

Windows 2000

to interconnect two or more subnets to form a larger multi-domain system area network. Within a subnet, each port is assigned a unique identifier by the subnet manager called the Local ID or LID. In addition to the LID, each port is assigned a globally unique identifier called the GID. Switches make use of the LIDs for routing packets from the source to the destination, whereas Routers make use of the GIDs for routing packets across domains. More detailed information on the LIDs, GIDs, and their assignment is available either in the specification or in William T. Futral's book (William T. Futral, *InfiniBand Architecture: Development and Deployment. A Strategic Guide to Server I/O Solutions*, Intel Press, 2001).

Performance Guide

By Mark Friedman, Odysseas

Pentakalos

[Table of Contents](#)

[Index](#)

[Sample Chapter](#)

[Full Description](#)

One more feature of the InfiniBand Architecture that is not available in the current shared bus I/O architecture is the ability to partition the ports within the fabric that can communicate with one another. This is useful for partitioning the available storage across one or more servers for management reasons and/or for security reasons.

Before I discuss a few more important benefits of the InfiniBand Architecture, we need to dig a little deeper. Figure 3 illustrates the communications stack of the InfiniBand Architecture. Before I go through each of the terms that appear in the figure, we need to understand the drivers that brought about the structure of the InfiniBand network stack.

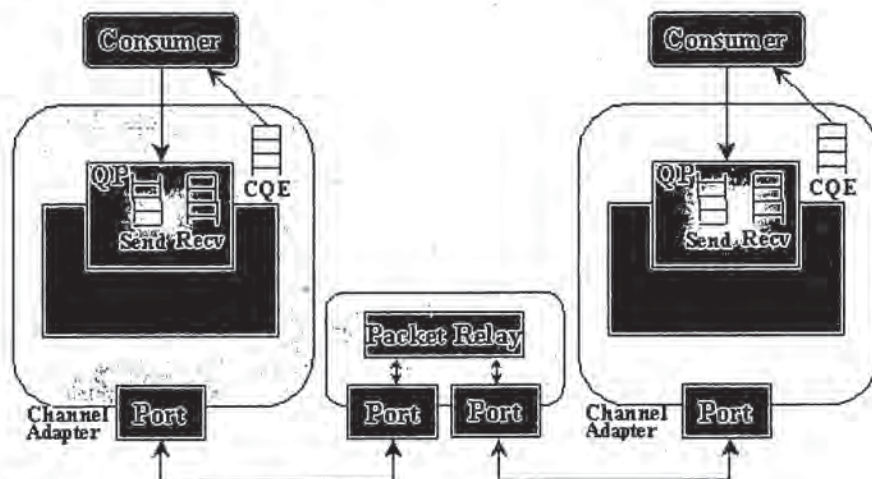


Figure 3. InfiniBand Communication Stack

In order to achieve better performance and scalability at a lower cost, system architects have come up with the concept of clustering, where two or more servers are connected together to form a single logical server. In order to achieve the most benefit from the clustering of multiple servers, the protocol used for communication between the physical servers must provide high bandwidth and low latency. Unfortunately, full-fledged network protocols such as TCP/IP, in order to achieve good performance across both LANs and WANs, have become so complex that both incur considerable latency and require many thousands of lines of code for their implementation.

To overcome these issues, Compaq, Intel, and Microsoft joined forces and came up with the Virtual

Interface (VI) Architecture (VIA) specification, which was released in December of 1997. The VI Architecture is a server messaging protocol whose focus is to provide a very low latency link between the communicating servers. The specification defines four basic components: virtual interfaces, completion queues, VI Provides, and VI Consumers. The VIA specification is available from [here](#), and it describes in detail each of the components. I won't describe them in detail here, and I only try to provide high-level information so that you can understand how low the latency is achieved.

In transferring a block of data from one server to another, latency arises in the form of overhead and delays that are added to the time needed to transfer the actual data. If we were to break down the latency into its components, the major contributors would be: a) the overhead of executing network protocol code within the operating system, b) context switches to move in and out of kernel mode to receive and send out the data, and c) excessive copying of data between the user level buffers and the NIC memory.

Since VIA was only intended to be used for communication across the physical servers of a cluster (in other words across high-bandwidth links with very high reliability), the specification can eliminate much of the standard network protocol code that deals with special cases. Also, because of the well-defined environment of operation, the message exchange protocol was defined to avoid kernel mode interaction and allow for access to the NIC from user mode. Finally, because of the direct access to the NIC, unnecessary copying of the data into kernel buffers was also eliminated since the user is able to directly transfer data from user-space to the NIC. In addition to the standard send/receive operations that are typically available in a networking library, the VIA provides Remote Direct Memory Access operations where the initiator of the operation specifies both the source and destination of a data transfer, resulting in zero-copy data transfers with minimum involvement of the CPUs.

Now the reason why I spent so much time talking about the VIA in an article about the InfiniBand Architecture is because the InfiniBand uses basically the VIA primitives for its operation at the transport layer. Now we can return to Figure 3 and describe all the terms that are shown. In order for an application to communicate with another application over the InfiniBand it must first create a work queue that consists of a queue pair (QP). In order for the application to execute an operation, it must place a work queue element (WQE) in the work queue. From there the operation is picked-up for execution by the channel adapter. Therefore, the Work Queue forms the communications medium between applications and the channel adapter, relieving the operating system from having to deal with this responsibility.

Each process may create one or more QPs for communications purposes with another application. Instead of having to arbitrate for the use of the single queue for the NIC card, as in a typical operating system, each queue pair has an associated context. Since both the protocol and the structures are all very clearly defined, queue pairs can be implemented in hardware, thereby off-loading most of the work from the CPU. Once a WQE has been processed properly, a completion queue element (CQE) is created and placed in the completion queue. The advantage of using the completion queue for notifying the caller of completed WQEs is because it reduces the interrupts that would be otherwise generated.

The list of operations supported by the InfiniBand architecture at the transport level for Send Queues are as follows:

1. Send/Receive: supports the typical send/receive operation where one node submits a message and another node receives that message. One difference between the implementation of the send/receive operation under the InfiniBand architecture and traditional networking protocols is that the InfiniBand defines the send/receive operations as operating against queue pairs.

2. **RDMA-Write:** this operation permits one node to write data directly into a memory buffer on a remote node. The remote node must of course have given appropriate access privileges to the node ahead of time and must have memory buffers already registered for remote access.
3. **RDMA-Read:** this operation permits one node to read data directly from the memory buffer of a remote node. The remote node must of course have given appropriate access privileges to the node ahead of time.
4. **RDMA Atomics:** this operation name actually refers to two different operations that have the same effect but which operate different from one another. The Compare & Swap operation allows a node to read a memory location and if its value is equal to a specified value, then a new value is written in that memory location. The Fetch Add atomic operation reads a value and returns it to the caller and then add a specified number to that value and saves it back at the same address.

For Receive Queue the only type of operation is:

1. **Post Receive Buffer:** identifies a buffer into which a client may send to or receive data from through a Send, RDMA-Write, RDMA-Read operation.

When a QP is created, the caller may associate with the QP one of five different transport service types. A process may create and use more than one QP, each of a different transport service type. The InfiniBand transport service types are:

- **Reliable Connection (RC):** reliable transfer of data between two entities.
- **Unreliable Connection (UC):** unreliable transfer of data between two entities. Like RC there are only two entities involved in the data transfer but message may be lost.
- **Reliable Datagram (RD):** the QP can send and receive messages from one or more QPs using a reliable datagram channel (RDC) between each pair of reliable datagram domains (RDDs).
- **Unreliable Datagram (UD):** the QP can send and receive messages from one or more QPs however the messages may get lost.
- **Raw Datagram:** the raw datagram is a data link layer service which provides the QP with the ability to send and receive raw datagram messages that are not interpreted.

This is not meant to be a complete coverage of the InfiniBand Architecture specification and as such I left lots of implementation details out. I do include a list of references though that should provide additional reading for those interested in learning more about it.

What state is the InfiniBand Architecture in and what is Microsoft doing about it?

Due to the overwhelming support in the industry for this new I/O standard architecture, InfiniBand development has been able to move very quickly from specification to actual products appearing in the market. Hardware for putting together InfiniBand fabrics started to appear in the 2nd quarter of 2001 and Beta OS Support was expected to be available in most popular operating system by the 4th quarter of 2001. Customer trials are expect after the 2nd quarter of 2002 with production systems starting to become available by the 3rd quarter of 2002.

At the Microsoft WinHEC 2001, Rob Haydt, Program Manager of the Windows Base OS Group, gave a presentation titled "Windows InfiniBand Support Roadmap" (Rob Haydt, "Windows InfiniBand Support Roadmap," WinHEC 2001.).

Related Reading



**Windows 2000
Performance Guide**

Within that presentation he indicated that InfiniBand will be supported in the first release of the Windows Whistler operating system. Second generation IB development will take place between late 2002-2003 and significant commercial deployments will begin until the 2003-2004 timeframe.

By Mark Friedman, Odysseas
Pentakalos
[Table of Contents](#)
[Index](#)
[Sample Chapter](#)
[Full Description](#)

If you are eager to try out some of the concepts described in this article but you don't have access to InfiniBand hardware, the easiest way to do that at this point is through Winsock Direct. Winsock Direct is an alternative implementation of the Winsock library that transparently takes advantage of support in the hardware to implement RDMA and kernel bypassing optimizations. It initially became available in Windows 2000 Datacenter Server but is now also available in the Service Pack 2 Release of Windows 2000 Advanced Server. The white paper by Jim Pinkerton that we have reference, describes in detail how Winsock Direct works and includes lots of numerical on experiments the author conducted so it makes for some interesting reading (Jim Pinkerton, "Winsock Direct: The Value of System Area Networks," Microsoft Corporation, May 2001). Other related interesting developments include the definition of the SCSI RDMA Protocol (SRP) over InfiniBand which is work in progress and the definition of the [Sockets Direct Protocol](#) (SDP) whose goal is to define a sockets type API over InfiniBand.

O'Reilly & Associates recently released (January 2002) [Windows 2000 Performance Guide](#).

- [Sample Chapter 5, Multiprocessing](#), is available free online.
- You can also look at the [Table of Contents](#), the [Index](#), and the [Full Description](#) of the book.
- For more information, or to order the book, [click here](#).

Odysseas Pentakalos has been an independent consultant for 10 years in performance modeling and tuning of computer systems and in object-oriented design and development.

[Return to the O'Reilly Network.](#)

Copyright © 2007 O'Reilly Media, Inc.

A Hierarchical Token Bucket Algorithm to Enhance QoS in IEEE 802.11: Proposal, Implementation and Evaluation

J.L. Valenzuela, A. Monleon, I. San Esteban, M. Portoles, O. Sallent
Dept. of Signal Theory and Communications - Polytechnic University of Catalonia (UPC)
Barcelona, Spain
valens@tsc.upc.es

Abstract—In this paper the performance of the IEEE 802.11 wireless local area networks in combination with Hierarchical Token Bucket traffic shaper is assessed. The analysis allows to show the basic advantages of the proposed scheduler. Results are obtained on a real IEEE 802.11b testbed to gain insight into HTB practical implementation issues. The HTB concept extended to a wireless scenario in this paper can also be applicable to IEEE 802.11e standard.

Keywords: WLAN, IEEE 802.11, QoS, throughput.

I. INTRODUCTION

Wireless LANs have experienced an impressive growth in the last years mainly by means of IEEE 802.11 family of standards, which has received the widest market acceptance. Besides, more complex services are envisaged to be supported in this competitive technology, such as multimedia communications, video on demand, VoIP, etc. create these components, incorporating the applicable criteria that follow.

Despite the important role of wireless LANs based in particular on 802.11b, it cannot be considered a well mature standard because of the performance, the lack of quality of service (QoS), security, etc. Nevertheless, the Task Group E within the IEEE 802.11 Working Group is developing a project (802.11e) whose purpose is to enhance the current 802.11 MAC to expand support for applications with Quality of Service (QoS) requirements [2]. In the meantime, and because of the plausible concerns about the time needed to stabilize and agree the standard in a first step and the time to market for 802.11e products in a second step, there have been proposals to provide QoS in 802.11b at MAC level, such as Distributed fair Scheduling [6], Blackburst [7], Aard's Differentiation Scheme [8], etc. These proposals modify parameters that define how a station (STA) access the wireless medium, i.e. they modify parameters of either the fundamental access method of the IEEE 802.11 MAC called Distributed Coordination Function (DCF), or the optional access method, which is the Point Coordination Function (PCF) [1]. In fact, some of these proposals are being included in the future Hybrid Coordination Function (HCF) in 802.11e [2].

The purpose of this paper is to provide appropriate quality of service mechanisms in WLAN with the novelty that, instead of focusing on MAC layer, the proposed solution is set at IP level with Hierarchical Token Bucket (HTB), which exercises

control over the transmissions, queuing and dequeuing packets in a determined and configurable way. HTB is a very complete and useful traffic shaper that has been successfully tested on wired environments [9] and [10]. This paper extends the use of this algorithm in a WLAN environment, and the proposed solution at IP layer could be incorporated into MAC layer as an enhancement to IEEE 802.11e.

One of the characteristics of current IEEE 802.11 products is the link adaptation, which consists on downgrade the bit rate transmission to a lower value when repeated unsuccessful frames transmissions are detected. This behaviour is shown to be very efficient for a standalone host. Nevertheless, link adaptation may seriously degrade the WLAN global performance, which penalizes fast hosts and privileges slow stations [3]. The HTB mechanism proposed in this paper can be a solution to the mentioned problem of stations transmitting at different rates. Furthermore, using Hierarchical Token Bucket less aggressive medium access behaviour can be achieved, with the corresponding positive influence on the throughput standard deviation.

Results for Hierarchical Token Bucket proposal in this paper have been obtained in a real WLAN testbed with IEEE 802.11b products. The testbed approach has been preferred in a first step in order to assess many practical issues regarding HTB in a real wireless environment. Since we found that HTB is a promising solution for future developments in combination with IEEE 802.11e enhancements, the next step would be to simulate this solution under IEEE 802.11e framework.

II. BASICS ON HIERARCHICAL TOKEN BUCKET

Hierarchical Token Bucket (HTB) is a class based queue discipline. A queue discipline (qdisc) can be seen as a black box which is able to queue and dequeue packets in order and at times determined by the algorithm hidden in it. It is located between IP layer and the layer 2 (MAC), as it is shown in figure 1.

HTB is based on hierarchical classes where three class types exist: root, inner and leaf. Root classes are suited on the top of the hierarchy and all traffic goes out through them. Inner classes have father and daughter classes. Finally, leaf classes are terminal classes, so they have father classes but not daughter classes. These three types of classes are described in

This work has been partly funded by the EVEREST project (IST-2002-001858) and by the Spanish Research Council under the grant TIC 2003-08609.

figure 2. In leaf classes, traffic from upper layers is injected following a classification which must be performed using filters, so it is possible to difference kinds of traffic and priorities, which should have different treatment. In this way, before traffic enters in a leaf class, it needs to be classified through filters with different rules, which can filter by kinds of services, IP addresses or even network addresses. This process is known as classifying process. Furthermore, when traffic has been classified, it is scheduled and shaped. In order to perform these tasks, HTB uses the concept of tokens and buckets to control the bandwidth use in a link. To adjust the throughput, HTB generates tokens at necessary cadence and dequeues packets from the bucket only if tokens are available. The main idea is shown in figure 3.

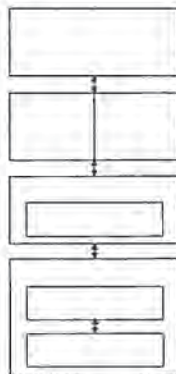


Figure 1. Layers detail.

The main advantage claimed for HTB traffic shaper is the bandwidth sharing. In this way, every class has associated an assured rate (AR), ceil rate (CR), actual rate (R), priority level (P) and Quantum (Q). This excess bandwidth is shared depending on the priorities that we have assigned to the classes. So, high priority classes can borrow more excess bandwidth than low priority classes. Thus, when R of one class has reached AR, it borrows tokens from its parent class. When this class has reached CR, it queues packets until new tokens/tokens are available. This process is known as policing process. The complete behavior is summarized in figure 4.

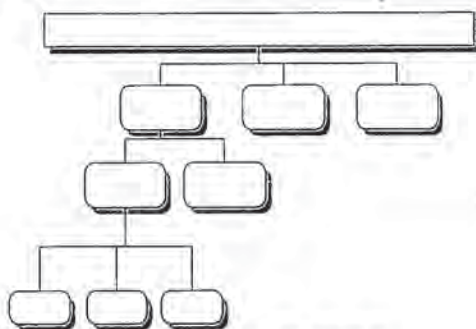


Figure 2. Hierarchical classes in HTB

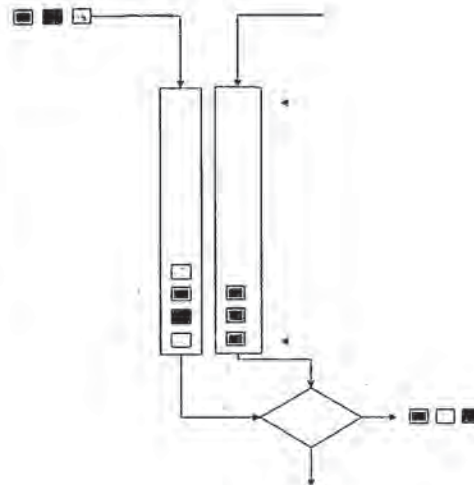


Figure 3. Token Bucket Filter performance.

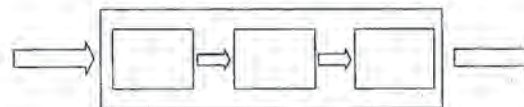


Figure 4. HTB summary behaviour

III. TESTBED IMPLEMENTATION

A common scenario as depicted in figure 5 have been considered. The server is connected to the Access Point via a 100 Mb/s switched Ethernet, so that the wireless link is the bottleneck. Furthermore, the application generated traffic for the test is only in the uplink, i.e. it goes from hosts to Access Point, because uplink traffic is more critical than downlink traffic. In addition, the wireless local network is configured in infrastructure mode, because of a relatively higher quality of service in this mode than in ad-hoc mode.

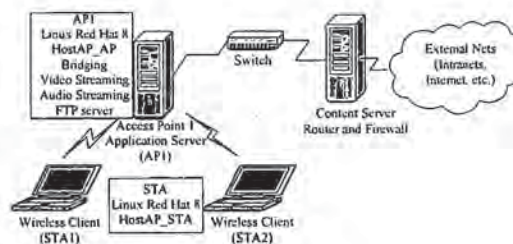


Figure 5. Scenario of tests: two hosts generate TCP and UDP traffic to a host behind the Access Point.

Current 802.11b products can degrade the bit rate from 11 Mbps to 5.5, 2 or 1 Mbps when a host detects repeated unsuccessful frame transmissions, in this scenario. To characterise these situation on the testbed, hosts have been

TCP

UDP

configured to send their packets at different fixed rates: 11Mbps, 11Mbps and 5.5Mbps, 2Mbps and 1Mbps respectively.

To generate the traffic we have used Iperf tool [12], which generates both TCP and UDP traffic, and ethereal tool [13], to capture all frames from client stations to content server and vice versa. HTB is included in Linux Kernel 2.4.20 and above.

IV. SAMPLE RESULTS

One very well known effect over IEEE 802.11 is that, in case of several host transmitting at different rates, the throughput of all hosts with higher rates are degraded below the level of the lower rate hosts as it is shown in figure 6. The testbed in this case has been configured with STA1 transmitting at 11Mbps, while STA2 transmits at 2Mbps. It can be observed that, instead of reaching the throughput of 5Mbps, it falls to a total throughput of 1.7Mbps approximately.

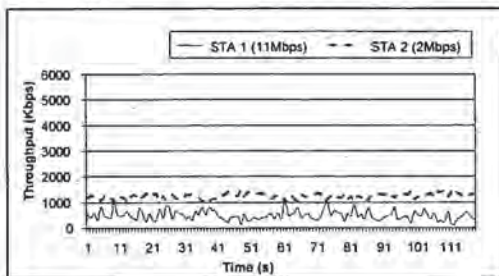


Figure 6. Two STA's transmitting at different rates.

The above result shows two basic problems: the first one is the average throughput of all hosts, imposed by the lowest bit rate host. The reason for this anomaly is the CSMA/CA channel access method of DCF, because it guarantees that the channel access probability is equal for all hosts. So, when a low rate host captures the channel, this station penalizes other hosts transmitting a longer time than fast hosts do [3]. For this reason, we need to find a solution that can deal traffic from different hosts independently, to guarantee a sustained throughput to all hosts. The second problem is the standard deviation, which reaches high values.

To cope with these situations, the traffic should be constrained by HTB in hosts in combination with the Access Point to obtain desired QoS effects. Performance using both UDP and TCP traffic are considered. In each experiment, the average throughput and standard deviation of both hosts are measured. The STA Bit rate column in the tables below shows the physical transmission rate, while HTB Limit column presents the assured rate configuration parameter of HTB algorithm. We group UDP and TCP results in terms of bit rate transmission due to the similar behaviour of the global system. The main difference between UDP and TCP tests is the total available throughput to share. In this way, TCP has less available bandwidth than UDP traffic, because TCP traffic needs ACK frames, and the access point competes for the channel too. As it is shown in [4], the proportion of useful

bandwidth depends on the number of hosts, and consequently, the useful bandwidth is lower for same conditions.

Following results are shown in two different parts, depending on the bit rate combination of both stations.

A. Two hosts transmitting at 11Mbps

The average throughput of each station in these tests is obtained. Then, HTB limits are configured with these values to analyse the new behaviour. As it can be seen, the average throughput results are very similar to the experiment without HTB, but the standard deviation has been significantly reduced. In fact, standard deviation is reduced from 156Kbps to 16Kbps for STA1 for UDP traffic and from 163Kbps to 13Kbps for STA2 for UDP. The standard deviation figures in Table I and Table II are complemented with figure 7 and figure 8, where it is shown the achieved throughput along time when HTB is not used and when HTB is included. The throughput stability obtained with HTB is clearly observed. Also notice that, in this case, where the two stations are transmitting at the same bit rate competing for the channel, throughput is shared in a linear way with 1:1 ratio and, consequently, STA1 and STA2 are obtaining the same throughput.

TABLE I.

HTB Status	STA Bit rate (Mbps)		HTB Limit (Kbps)		Average Throughput (Kbps)		Std Deviation σ (Kbps)	
	STA1	STA2	STA1	STA2	STA1	STA2	STA1	STA2
Without HTB	11	11	-	-	3018	3046	156	163
With HTB	11	11	3000	3000	3003	3000	16	13

Measured Throughput and Standard Deviation for 11 Mbps rates and UDP Traffic.

TABLE II.

HTB Status	STA Bit rate (Mbps)		HTB Limit (Kbps)		Average Throughput (Kbps)		Std Deviation σ (Kbps)	
	STA1	STA2	STA1	STA2	STA1	STA2	STA1	STA2
Without HTB	11	11	-	-	2546	2449	204	198
With HTB	11	11	2400	2400	2388	2397	50	35

Measured Throughput and Standard Deviation for 11 Mbps rates and TCP Traffic.

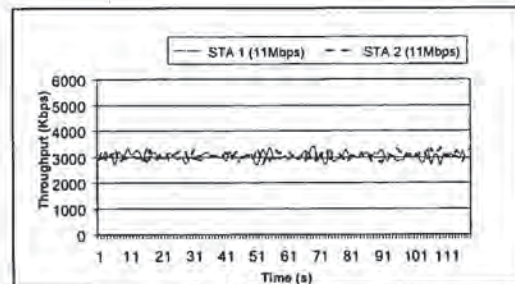


Figure 7. Throughput of two stations at 11Mbps without HTB.

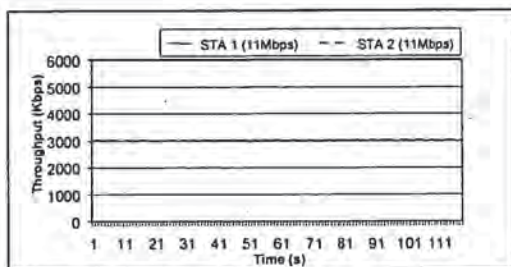


Figure 8. Throughput of two stations at 11Mbps with HTB.

B. One host transmitting at 11Mbps and another at 2Mbps

In this test, different bit rate transmissions are considered. STA1 is 11 Mbps bit rate and STA2 is 2Mbps. When STA2 transmits UDP traffic at 2 Mbps, it uses five longer time than STA1. For this reason, if HTB constricts STA2, available bandwidth to share is almost four times larger the bandwidth HTB has constricted. For example, when HTB limit of STA2 is reduced from 1300 Kbps to 900 Kbps, STA1 HTB limit is upgraded from 900 Kbps to 2400Kbps, so a 50 % gain on available throughput follows. Additionally, the standard deviation reduces significantly (Figure 9 and figure 10). In TCP traffic the gain is almost 53 %.

TABLE III.

HTB Status	STA Bit rate (Mbps)		HTB Limit (Kbps)		Average Throughput (Kbps)		Std Deviation σ (Kbps)	
	STA1	STA2	STA1	STA2	STA1	STA2	STA1	STA2
Without HTB	11	2	-	-	965	1303	164	62
With HTB	11	2	900	1300	900	1301	42	27
	11	2	2400	900	2404	901	56	11

Measured Throughput and Standard Deviation for 11 and 2 Mbps rates and UDP Traffic

TABLE IV.

HTB Status	STA Bit rate (Mbps)		HTB Limit (Kbps)		Average Throughput (Kbps)		Std Deviation σ (Kbps)	
	STA1	STA2	STA1	STA2	STA1	STA2	STA1	STA2
Without HTB	11	2	-	-	491	1233	201	103
With HTB	11	2	500	1200	477	1107	59	57
	11	2	1900	700	1873	719	56	19

Measured Throughput and Standard Deviation for 11 and 2 Mbps rates and TCP Traffic

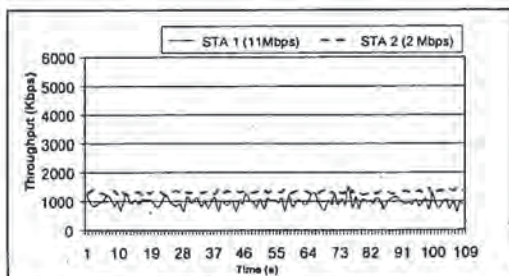


Figure 9. Throughput of two stations at 1 and 2Mbps without HTB

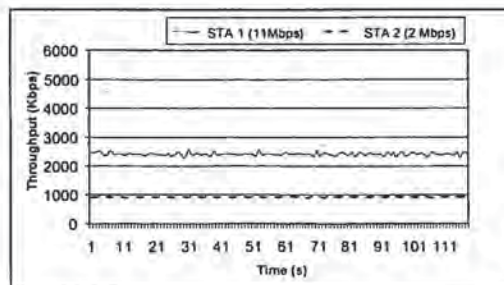


Figure 10. Throughput of two stations at 1 and 2Mbps with HTB

V. CONCLUSIONS

This paper has presented the performance of the IEEE 802.11b and it has described the well known problems of this technology, but specially the lack of quality of service. We propose the use of Hierarchical Token Bucket algorithm, located in the IP layer. This solution controls how the packets are delivered to MAC layer. This method is independent of the wireless standard used and it can be used jointly with any actual of future devices.

We can offer quality of service in wireless local area networks constraining stations transmitting at a lower bit rate. Applying this concept we can achieve a plain and sustained throughput with low standard deviation to stations or even to differentiated services.

REFERENCES

- [1] ANSI/IEEE, 802.11-1999, "Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications", IEEE.
- [2] IEEE 802.11 WG, "Draft Supplement to International Standard for Information Technology-Telecommunications and Information Exchange between Systems LAN/MAN Specific Requirements." IEEE 802.11e/D4.0, 2002
- [3] Martin Heusse, Franck Rousseau, Gilles Berger-Sabbatel, Andrzej Duda, "Performance anomaly of 802.11b", INFOCOM 2003.
- [4] J. Antonio Garcia-Macias, Franck Rousseau, Gilles Berger-Sabbatel, Leyla Toumi, Andrzej Duda, "Quality of Service and Mobility for the Wireless Internet", Wireless Networks 9, 341-352, 2003.
- [5] Yasir Drabu, "A Survey of QoS Techniques in 802.11", 2002. Available in <http://www.cs.kent.edu/~ydrabu/>
- [6] Nitin H. Vaidya, Paramvir Bahl, Seema Gupta, "Distributed fair scheduling in a wireless LAN," Proceedings of the sixth annual international conference on Mobile computing and networking, p.167-178, August 06-11, 2000, Boston, Massachusetts, United States.
- [7] J.L. Sobrinho and A.S. Krishnakumar, "Real-time traffic over the IEEE 802.11 medium access control layer," Bell Labs Technical Journal, pages 172-187, Autumn 1996.
- [8] I. Aad, C. Castelluccia, "Differentiation mechanisms for IEEE 802.11," Proceedings of IEEE INFOCOM 2001, vol. 1, pp. 209-218, April 2001.
- [9] Martin Devera, "Hierarchical Token Bucket", <http://luxik.cdi.cz/~devik/qos/htb/>
- [10] Bert Hubert, Gregory Maxwell, Remco van Mook, Martijn van Oosterhout, Paul B Schroeder, Jasper Spaans, Pedro Larroy, "Linux Advanced Routing & Traffic Control HOWTO", <http://lartc.org/howto/>
- [11] Martin A. Brown, "Traffic Control HOWTO", <http://www.linuxsecurity.com/docs/LDP/Traffic-Control11QWTO/>
- [12] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, Kevin Gibbs, "lperf", <http://dasi.nlanr.net/Projects/lperf/>
- [13] Gerald Combs, "Ethereal", <http://www.ethereal.com/>



Evaluating high speed industry standard serial interconnects

By Harpinder S. Matharu

Decision criteria

What performance capabilities should an industry standard switching fabric interconnect offer? Ideally, such technology should support bandwidth that exceeds the aggregate of all line cards in the system including protocol overhead, and should be capable of transporting control and data information across all the diverse entities constituting the system. In addition, latency and jitter should be minimal.

The desire for higher interconnect speeds between chips, boards, and chassis continues to grow in order to satisfy the requirement for higher throughput in future systems. It is now apparent that parallel shared-bus architectures no longer provide a viable option.

To address these requirements and challenges, some communication equipment manufacturers have turned to proprietary high-speed interconnects. However, these are short term solutions due to their high cost, long-lead time, and lack of interoperability between proprietary interconnects.

Over the long term, equipment designers need industry-standard solutions capable of sustaining high speed data transfer at low cost. Based on earlier communication standards, new serial interconnect technologies such as InfiniBand, RapidFabric, and Advanced Switching Interconnect (ASI) deliver an industry-standard approach. This approach allows equipment designers to reuse software and maintain backward compatibility with legacy equipment while leveraging the cost efficiencies of the existing infrastructure.

In this article, Harpinder first describes each of the three architectures. He then compares each of these three competing architectures by focusing on their flow control, congestion management, and high availability characteristics.

An interconnect fabric must also provide flow control capabilities to ensure a high Quality of Service (QoS) level. To achieve this goal, the system must provide congestion management policies and mechanisms capable of preventing an overload of the link and component capacities within the fabric. In addition, the fabric must provide the right balance of capabilities to support diverse applications without driving up device complexity and cost.

Reprinted from Embedded Computing Design / July 2005

The switch fabric should also be protocol agnostic so as to deliver a high degree of flexibility for the backplane interconnect. Its underlying topology should be designed with redundancy and failover mechanisms that support the high availability, serviceability, and reliability requirements of communications equipment.

Finally, scalability, flexibility, and extensibility are also key features in any high speed serial interconnect architecture. Ideally, the hardware should support these features and simplify deployment of fabric management software. The software should be user friendly, inexpensive, portable, extensible, and should allow centralized or distributed fabric management to support load balancing and system redundancy.

can be as simple as an Ethernet adapter or as complex as a high performance computing blade. The specification defines the hardware protocol for reliable message transport but does not define the content of the message. Hardware protocols allow data transfer from both the kernel and user spaces of the operating system.

The architecture supports a variety of fabric services ranging from configuration and asset management and error reporting to performance metric collection and topology management. It natively uses IPv6 headers to more efficiently exchange data between the InfiniBand architecture fabrics and the Internet.

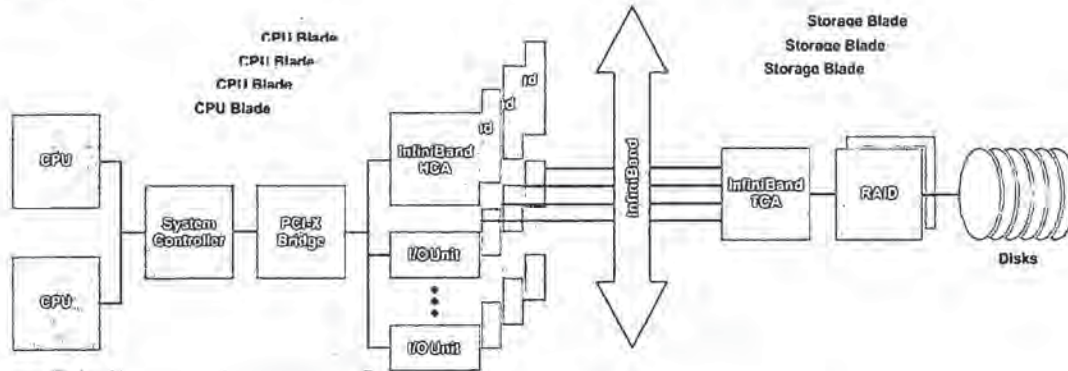


Figure 1

Architectural overviews

InfiniBand architecture

The oldest of the three discussed technologies, the InfiniBand Architecture (IBA) is a comprehensive serial interconnect specification. It was designed from the ground up to be optimized for storage area networking, high end computer clustering, and local area networking applications that require high availability and reliability. An InfiniBand server blade cluster with storage system is shown in Figure 1.

InfiniBand architecture has the following characteristics:

- Has a clock speed of 2.5 Gbps
- Supports x1, x4, and x12 lanes
- Supports throughputs of 2, 8, and 24 Gbps
- Supports module-to-module interconnects
- Supports chassis-to-chassis interconnects
- Supports QoS with 16 levels of virtual channels that are mapped to 16 service levels

InfiniBand is implemented with a four-layer stack with the following layers:

- Link
- Network
- Transport
- Physical

Devices in the system are identified by local, global, and EUI-64 addresses. Transport services provide both reliable and unreliable data delivery. Payload size can range up to 4096 bytes with up to 126 bytes of header overhead.

The architecture provides a reliable transport mechanism using messages between end nodes. Nodes in an InfiniBand configuration

RapidFabric architecture

Over the last two years, the RapidIO Trade Association has worked to extend the Serial RapidIO (SRIO) interconnect to cover data plane applications for telecommunications networks. The original SRIO specification supplies a simple chip-to-chip serial interconnect for processor interconnect applications and is widely used today in DSP arrays.

Designed to replace proprietary data fabrics with an open standards technology, RapidFabric adds multicast, flow control, and data streaming features to the base SRIO specification. A RapidFabric system is shown in Figure 2.

RapidFabric architecture has the following characteristics:

- Has a clock speed of 3.125 Gbps
- Supports x1 and x4 lanes
- Supports throughputs of 1.25, 2.5, and 3.125 Gbps
- Supports a bandwidth of 10 Gbps with x4 lanes

RapidFabric is implemented with a three-layer stack with the following layers:

- Logical
- Transport
- Physical

The logical layer defines the protocols utilized by the endpoints to carry out interconnect services. The transport layer defines the addressing scheme for delivery. Broadcasting and multicasting capabilities are implemented by manipulating the transport information. The physical layer defines the packet transport mechanism, flow control, and electrical characteristics. It supports up to three transport flows and four virtual channels.

Devices in a RapidFabric configuration are connected by device-based routing. The architecture supports the interconnection of

Reprinted from *Embedded Computing Design* / July 2005

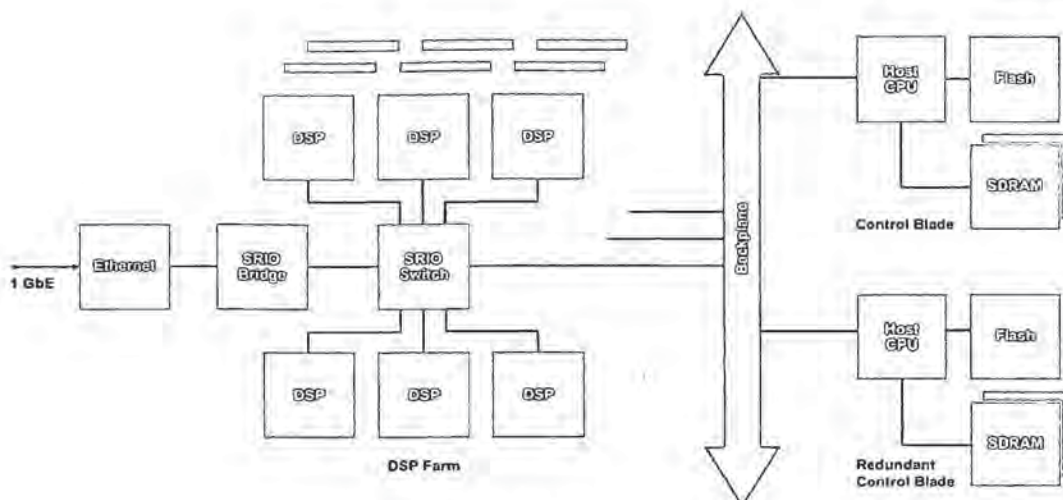


Figure 2

up to 64K devices with centrally managed routing tables. Packet headers are small. Payload size is limited to 256 bytes, but raw data streaming allows transfer of up to 64 kilobytes.

ASI architecture

Leveraging the industry's extensive investment in PCI Express (also denoted as PCIe) equipment, the ASI architecture offers developers a foundation for implementing multipoint, peer-to-peer switched interconnect links for both data plane and control plane communications. An ASI system is shown in Figure 3.

ASI architecture has the following characteristics:

- Has a clock speed of 2.5 Gbps
- Supports x1, x2, x4, x8, x12, x16, and x32 lanes
- Supports throughputs up to 64 Gbps

ASI is implemented with a three-layer stack with the following layers:

- Data Link
- Transaction
- Physical

Both the physical and data link layers replicate those used in PCIe with minor enhancements. However, the transaction layer in ASI has been completely rewritten to support the needs of a backplane interconnect.

Maximum packet size is 2176 bytes, but the architecture can support unlimited size packets through the use of a native Segmentation And Reassembly (SAR) capability. It features path-based routing and a reliable transport mechanism with an option for unreliable delivery. The ASI packet encapsulation scheme offers low header overhead and tunnels any protocol. It provides deterministic behavior through low latency and good jitter control.

The ASI architecture provides twenty virtual channels divided into three types:

- Eight bypass capable channels are used to transport load-store protocols by incorporating mechanisms to prevent potential deadlocks

- Eight ordered-only channels are used for message oriented push data traffic
- Four multicast virtual channels are used to host any application profile

At the logical level, the standard supports eight traffic classes per virtual channel type for QoS and traffic differentiation. Primitives defined in the specification support high availability capabilities such as hot insertion/removal, redundant paths, multiple owner entities, and efficient fabric management failover.

ASI implements 127 Protocol Interfaces (PIs) for tunneling protocols and fabric management. The standard defines protocol interfaces for:

- Simple Queue (SQ)
- Simple Load Store (SLS)
- PCI Express bridging (PI-8)
- Sockets Data Transfer (SDT)

In a typical ASI fabric topology, an I/O device or CPU can natively support an ASI interface or connect to the ASI fabric using an ASI bridging device. This allows the PCIe tree hierarchy to span multiple CPU domains by using host switches and I/O switches. The host switch allows CPUs to connect to diverse I/O devices made available in the fabric through I/O switches.

CPUs can exchange data using SQ, SLS, and SDT protocol interfaces without the need to traverse the higher layer protocol stack (such as TCP). This also allows data traffic received from an ingress NPU card to be bridged to the ASI fabric using a PI-SPI protocol interface and eventually routed to an egress NPU card.

Flow control

InfiniBand flow control

Flow control capabilities play a crucial role in each standard's ability to deliver a high QoS level. With the InfiniBand architecture, flow control is built around a credit-based scheme. When the transmitter channel adapter operates at a higher link speed than the destination ingress link speed, the intermediate switch provides *back pressure* to the transmitting channel adapter by reducing the link-level flow control credits. This function prevents the system from overrunning the resources of the slower destination device.

Reprinted from *Embedded Computing Design* / July 2005

InfiniBand uses Virtual Lanes (VLs) to implement logical flows over a single physical link. Each IBA device supports up to 16 VLs. VL 15 and VL 0 are implemented in all devices. VL 15 is used for subnet management, while VL 0 is provided for application use. Internal device buffers are allocated separately for each VL. Each device maps a Service Level (SL) to a VL. Packets are ordered for the same source and destination Local IDentifiers (LIDs) or address and SL. The system does not provide any bypass capability.

VL arbitration per port assigns the highest priority to VL 15 and the second highest priority to flow control packets. A two-level scheduling mechanism is used for all other VLs. A pre-emptive scheduler overlaid on top of a weighted fair scheme ensures fairness while allowing progress on lower priority VLs. The weighting priority and minimum forward progress bandwidth is programmable. An arbiter uses *Limit of High Priority* to indicate the number of high-priority packets that the system can transfer before a lower priority packet can be transmitted. The flow control packets send credit information to the transmitter per VL at a periodic rate.

RapidFabric flow control

In comparison to InfiniBand, RapidFabric implements a much simpler flow control mechanism. The physical layer in RapidFabric implements a 2-bit field in each packet that is used to assign priority. The logical layer defines any of three types of transaction requests (highest, medium, or lowest) that are mapped to one of the priorities at the physical layer.

Flow control in this scheme can be controlled by the receiver or the transmitter. When the receiver controls the flow, it keeps accepting ingress data on a packet-by-packet basis depending on buffer availability. If buffer space is not available, it drops the received packets and informs the transmitter by sending a packet-retry control symbol. The transmitter then waits or retransmits the packet. The receiving fabric generates flow control packets to start (XON) and stop (XOFF) the flow of traffic to it from a specific source fabric element.

When data flow is controlled by the transmitter, the system relies on a credit-based scheme. In this case, the receiver informs the source periodically how much buffer space is available and the transmitter regulates data flow accordingly.

ASI flow control

By contrast, the ASI architecture uses a credit-based flow control mechanism between peer entities per each virtual channel. A source fabric element in ASI can transmit a packet to its peer only when enough credits are available.

The ASI architecture provides twenty virtual channels divided into three types:

- Eight virtual channels are unicast, Bypass-capable Virtual Channels (BVCs) channels designed for the transport of load-store protocols. These channels can carry ordered as well as by-passable data traffic. Separate credits are allocated for the two traffic types to regulate their flow through the fabric.
- Eight channels are unicast only.
- Four channels support multicast operation.

Data traffic is differentiated and isolated through the use of different class identifiers, a virtual channel queuing mechanism, and egress link scheduling.

Peer entities determine the number of unicast BVC and Ordered-only Virtual Channels (OVCs) that each supports as part of the initial exchange of virtual channel flow control credit information. If one link partner supports more BVCs than the other, then the specification allows transformation of excess BVCs into OVCs.

ASI was also designed to support cut-through traffic. This capability allows a switch to forward a packet before it is completely received. In order to comply with credit-based flow control, the ASI route header adds credit required information for each packet. The switch reads the *credits required* field of the received packet to determine whether it has enough credits to forward it before completely receiving the entire packet.

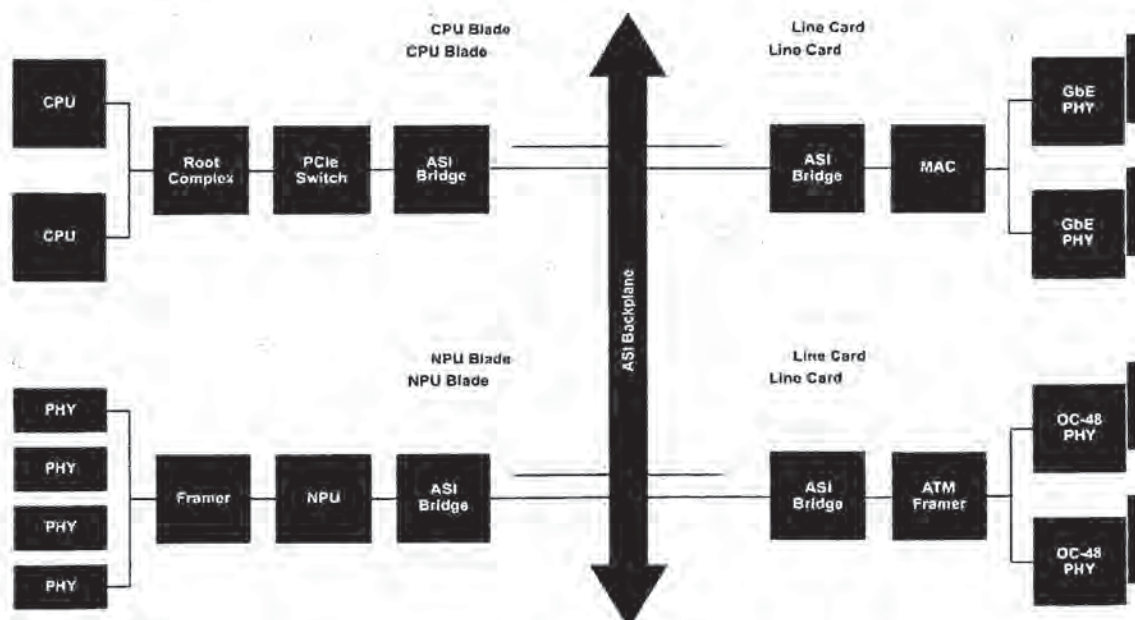


Figure 3

"The physical layer in RapidFabric implements a 2-bit field in each packet that is used to assign priority. The logical layer defines any of three types of transaction requests ... that are mapped to one of the priorities at the physical layer."

Congestion management

InfiniBand congestion management

The InfiniBand architecture uses a static rate control mechanism to reduce the data rate sourcing speed of an end node into the fabric. Static rate algorithms control the programmable interpacket delay between packets emerging from an end point and going to the same destination.

This preconfigured value is determined by the device provided port-rate information, or by using a fabric manager-based database of best possible rates for a source destination pair. Congestion management in InfiniBand does not support any dynamic rate control schemes to allow high speed data bursts.

RapidFabric congestion management

Congestion management in RapidFabric is based on simple XON/XOFF controls on transaction request flows. Fabric elements track their internal packet buffer levels on a packet-by-packet basis that corresponds to a programmable and locally defined watermark level. If a packet-buffer level exceeds the watermark level, the receiver sends an XOFF control message to shut off the source.

Watermark levels are determined by a number of factors including a fabric element's location in the fabric, its distance from the source, and the fabric topology. They should allow enough buffer space in the fabric element for storing packets-in-flight while taking into account the worst-case XOFF latencies. When a low watermark is passed, the receiver sends an XON packet to the source to resume transmission.

ASI congestion management

ASI offers more comprehensive congestion management functions than the other two serial interface architectures. To avoid traffic congestion based on virtual channels, ASI adds three additional mechanisms:

- Status-Based Flow Control (SBFC)
- a minimum bandwidth scheduler
- end-point source or injection-rate limiting

Under SBFC, a node in the system passes status Data Layer Link Packets (DLLP) that contain information about the buffer space available at the switch to its upstream neighbor. Depending upon the status of available buffer space in the path to destination, the upstream device egress scheduler starts or temporarily suspends any flow identified by the status DLLP.

This proactive mechanism allows the system to avoid any situation where the devices run out of credits to transmit. The minimum

bandwidth or vendor-defined egress scheduler can be used by devices to throttle traffic in accordance with indicated Traffic Class/Virtual Channel (TC/VC) mappings.

ASI also supports optional source rate limiting by sorting packets into connection queues based upon criteria such as traffic class or packet path. A token bucket can be paired with each connection queue to provide source rate limiting. Token buckets provide an admission control limit to the average transmission rate per connection queue while allowing transmission of controlled bursts through the fabric. If congestion still occurs in the fabric despite these controls, it is important to quickly throttle the congesting source end point. The source can be identified by the route header.

High availability

High availability is a crucial consideration for any communications fabric. Interconnect error detection and error handling capabilities play a key role in a system's ability to implement highly efficient failover solutions.

InfiniBand availability

InfiniBand, initially designed with clustering and storage area networking in mind, offers good capabilities in this area. The error detection and handling mechanism differs at the requester and the responder. All error events are handled either at the requester or responder or at both locations.

At the requester, events can be either local or remote. Both types of errors are entered into the completion queues, which in turn invoke an event handler.

At the responder, the device can silently discard a packet, send back the error in the acknowledge packet, queue the error in the completion queue, or use a combination of all three options. A subnet manager periodically scans the fabric for changes. Alternately, the event handler can update the subnet manager.

RapidFabric availability

RapidFabric relies heavily on capabilities embedded in the RapidIO specification. To improve performance while keeping the design simple, RapidIO avoids regenerating the Cyclic Redundancy Check (CRC) as the packet moves through the fabric. However, the CRC check is performed at every hop.

If a packet is received with a bad CRC or format error, then the receiver informs the sender of the error and enters an *Input Error-stopped* condition and silently discards all new packets until it receives a *restart-from-error* message from the sender. Link maintenance control symbols are used to coordinate with the sender.

The system uses response time-out counters to detect packet loss. The port response time-out is programmable. Port error and Command and Status Registers (CSRs) indicate error conditions, which need to be reset by the software error handler.

ASI availability

ASI devotes a specific Protocol Interface (PI-5) for event handling and management. An event in ASI can be handled locally or returned to the sender. The specification also allows routing of events to a dedicated event manager.

ASI uses the link layer, transaction layer header, and payload CRCs to perform packet integrity checks. The link layer CRC is transparently generated and checked in the data link layer of the link partners. The header CRC is checked at every hop along the

Reprinted from Embedded Computing Design / July 2005

**"If a packet is received
with a bad CRC or format error,
then the receiver informs the
sender of the error and enters an
Input Error-stopped condition and
silently discards all new packets until it
receives a *restart-from-error*
message from the sender."**

path. The Payload CRC (PCRC) is generated by the source end point and checked at the receiver end point. It provides end-to-end integrity checks and error notification.

Switch elements that forward packets do not have to implement PCRC. In addition, ASI packets use a header CRC that can be checked on a per hop basis by switches as they forward packets. If a header CRC error is detected, the packet is discarded at an intermediate switch or at the terminus of the packet and an ASI route header error is signaled.

Summary

A summary of the comparison of the three fabric standards is shown in Table 1 (next page). Each of the new standard serial

interconnects offers developers a unique set of capabilities. The distinct characteristics of protocol support, flow control, congestion management, and error detection and event handling offer designers advantages for different applications.

System designers must also consider each technology's ability to exploit its underlying infrastructure and leverage the economies of scale before they select a solution for their application. The ecosystem surrounding a given technology will have a profound impact not only on its end cost, but on the speed of its eventual adoption. For example, the flourishing PCI Express ecosystem may make Advanced Switching (ASI) an attractive choice for a large number of applications. ECD

Harpinder S. Matharu, Principal Engineer, IDT Inc. Harpinder has more than 15 years of experience as a software and hardware engineer in communication systems design. He holds a Bachelor of Engineering degree in Electronics and Communication from the Delhi Institute of Technology, and a Master in Business Administration degree from the University of Phoenix.



For more information, contact Harpinder at:

IDT Inc.
2975 Stender Way • Santa Clara, CA 95054
Tel: 408-725-6116 • Fax: 408-492-8264
E-mail: Harpinder.Singh@idt.com
Website: www.idt.com

	InfiniBand	SRIO/RapidFabric	ASI
Maximum bandwidth (after 8B/10B encoding overhead)	24 Gbps	10 Gbps	64 Gbps
Lanes supported	x1, x4, x12	x1, x4	x1, x2, x4, x8, x12, x16, x32
PHY speed	2.5 Gbps	1.25 / 2.5 / 3.125 Gbps	2.5 Gbps
Packet types	Reliable connection Reliable datagram Unreliable connection Unreliable datagram Raw IPv6 datagram Raw Ethernet packets	Read Write Sequenced write Atomic operations Maintenance operations Messages Data streams	Tunnels any packet and any protocol Simple Queuing (SQ) Simple Load/Store (SLS) Sockets Direct Transfer (SDT) Multicast Data Streaming and SAR Management packets (PI-4: device management, PI-5: event management, PI-1: congestion management)
Data movement model	Datagram, Streams, RDMA	Load/Store, Messaging	Load/Store, Messaging, Queuing, RDMA
Maximum payload	4096 bytes	256 bytes	2176 bytes
Transaction layer overhead	High, header overhead as high as 126 bytes	Small, 12 to 16 bytes	Small, 12 or 16 bytes
Reliable transport	Yes	Yes	Yes
Multi-cast	Yes	Yes	Yes
Routing	Device-based routing	Device-based routing	Path-based routing
Class of Service	16 Service Levels or Traffic Classes mapped to 16 Virtual Channels	Three transaction requests mapped into four priorities. RapidFabric enhancements have added more classes of service.	Eight Traffic Classes mapped each to eight Bypass capable, eight ordered, and four multicast independently buffered Virtual Channels.
Flow control	Credit-based, Static rate control	XON/XOFF	Credit-based, Status-based (SBFC), minimum bandwidth scheduler, injection rate limiter.
Error detection	Variant and Invariant CRC	CRC (no regeneration, per hop checking)	Header, Link, and End-to-End CRC
Packet forwarding	Does not provide header CRC for cut-through forwarding. Opportunistic cut-through forwarding could be supported.	Allows cut-through only after checking CRC (16-bit) on receiving 82 bytes of the packet. It is not supported for packet size of 80 bytes or less.	Allows cut-through forwarding after checking Header CRC (7-bit) on receiving eight bytes of the packet.
Fabric management	Multiple Host Channel Adapters (HCAs) competing per subnet. SNMP extension.	Multiple hosts compete	Fabric owner election process via spanning tree.

Table 1

An Overview of QoS Capabilities in InfiniBand, Advanced Switching Interconnect, and Ethernet

Sven-Arne Reinemo, Tor Skeie, Thomas Sødning, and Olav Lysne, *Simula Research Laboratory*
Ola Tørudbakken, *Sun Microsystems*

ABSTRACT

A recent trend in interconnection network technologies is the inclusion of various mechanisms to support a variety of quality of service (QoS) concepts. This has been necessitated by an increasing number of application areas that require some level of performance guarantees from the network for parts of its traffic. In this article we describe and compare the capabilities and support for the QoS of three of the most important interconnection network technology standards of today. Equalities between the technologies are explained and differences are clarified.

INTRODUCTION

Research into the quality of service (QoS) in interconnection networks has gone through several phases. Two decades ago interconnection networks with point-to-point links were mainly used in massively parallel processors undertaking scientific calculations. At that time the interconnection network was considered a bottleneck in the computation and therefore most of the research and development effort focused on improving overall performance, while differentiation between different classes of traffic did not receive noteworthy attention. This resulted in rapid increases in interconnection performance and when new application areas related to multimedia and other real-time applications came along, the bandwidth of the interconnection network was widely believed to be abundant. Taking special measures to control the QoS was therefore considered unnecessary. It was not until the late 1990s that this view changed in a profound way, and at present there is an expectation that QoS-controlling mechanisms are part of any general-purpose interconnection standard.

When discussing QoS in interconnection networks, there are three properties of significant importance: *bandwidth*, *latency*, and *packet loss*. The granularity of the object on which these metrics are applied are single data streams, classes of traffic, or all-network traffic. In most

interconnection technologies, there is a strict guarantee of no packet loss that is valid for all data traffic. Ethernet can be viewed as an exception to this, which we discuss below. With regard to latency and bandwidth, a combination of the mechanisms are often defined, ranging from strict guarantees for single streams [1] via relative guarantees for classes of traffic [2] to no guarantees/overprovisioning.

The capabilities that influence the technologies' ability to leverage QoS guarantees and differentiated treatment of traffic fall into three categories: flow-control, congestion management, and traffic differentiation.

Flow-control aims to reduce or eliminate packet loss that is a result of contention and overflowing receive-buffers in switches; however, its use can result in congestion in one part of the network spreading to others parts as link transfers slow down or are temporarily halted. This phenomenon is known as back-pressure. Congestion management aims to prevent or react to the onset of congestion, reducing or controlling its effect on overall throughput in the network. Traffic differentiation applies differential treatment to traffic in order to provide certain guarantees to particular streams. A complex application (e.g., a video server) deals with a multitude of traffic types, each with different requirements for timely delivery. Some of the traffic may be sensitive to delay but without strict bandwidth requirements, for example, network control and management traffic. Other types of traffic (e.g., video streams) may have strict bandwidth requirements whereas the latency requirements are relaxed.

In this article we present an overview and comparison of flow-control, congestion management, and traffic differentiation mechanisms in the InfiniBand Architecture (IBA) [3], Advanced Switching Interconnect (ASI) [4], and what has so far been specified for Backplane Ethernet (by the IEEE 802.3ap Backplane Ethernet Task Force). We also clarify the situations where the technologies give different names and descriptions for what is essentially the same mechanism.

This work is financed in part by the EU under the 6th framework program for IST as part of the SIVSS project. Sven-Arne Reinemo, Tor Skeie, and Thomas Sødning are all primary authors.

Feature	IBA	ASI	Ethernet
Routing	Destination lookup	Source routing	Destination lookup
Link width (no. of lanes)	1x, 4x, 12x	1x, 2x, 8x, 16x, 32x	1x, 4x (for 3.125 Gb/s)
Bandwidth per lane	2.5, 5, 10 Gb/s	2.5, 5 Gb/s	1, 3.125, 10 Gb/s
Bandwidth	2.5–120 Gb/s	2.5–128 Gb/s	1–10 Gb/s
Maximum packet size	4096 bytes	2176 bytes	1522 bytes (9000 bytes is supported by many vendors)
Minimum packet size	24 bytes (20 bytes raw)	64 bytes	64 bytes
Transmission encoding	8B/10B	8B/10B	8B/10B, 64/66B for 10 Gb/s
Maximum cable length	Unspecified	Unspecified	5000 m
Maximum number of hosts	49,152	Unspecified	Unspecified
Maximum ports per switch	255	256	Unspecified

■ Table 1. IBA, ASI, and Ethernet features overview.

The rest of the article is organized as follows. First, we present a basic description of each of the three technologies under review. We then detail the technologies' ability to support flow-control, congestion management, and service differentiation. Thereafter, we discuss the mechanisms provided and how they support the higher-level properties of QoS that have motivated the development of these mechanisms before we conclude in the final section.

ARCHITECTURAL OVERVIEW

INFINIBAND ARCHITECTURE

The InfiniBand Architecture (IBA) was first standardized in October 2000 [3], as a merging of two older technologies called Future I/O and Next Generation I/O. As with most other recent interconnects, IBA is a serial point-to-point full-duplex interconnect. It was originally designed as a unified I/O fabric to replace everything from the PCI bus inside commodity servers and FibreChannel in storage area networks, to Ethernet in system area networks. While it has not been successful in achieving this, it has found its niche as an *intersystem* interconnect in storage and high-performance computing, where high-bandwidth and low-latency networks are key requirements. The feature set of each technology is presented in Table 1.

ADVANCED SWITCHING INTERCONNECT

Since its inception in the mid-1990s the PCI interconnect has evolved from a parallel, bus-based technology to a serial, switched one. In its latest incarnation, PCI Express IO v1.0 is a low latency, high-bandwidth *intrasystem* interconnect [5]. Advanced Switching Interconnect (ASI) is an *intersystem* interconnect that is built upon the same technology as PCI Express, reusing the same physical and link layers but with its own transaction layer to provide peer-to-peer communication within an ASI fabric [4].

ETHERNET

Since its invention at Xerox PARC in 1973, local area networking (LAN) has been, by far, the most important application domain for Ethernet technology [6, 7]. Today, Ethernet is the dominant LAN technology for both wired and wireless networking but is now making inroads into new application areas ranging from clustering and storage networks to wide area networking. Furthermore, Ethernet is currently undergoing a standardization process for the backplane in the IEEE 802.3ap Backplane Ethernet Task Force.

ELEMENTS OF QUALITY OF SERVICE

In the following we assess the flow-control, congestion-management, and differentiated-services mechanisms for IBA, ASI, and Ethernet, before we resume a discussion regarding the QoS capabilities of each of the technologies. The different terms used by each technology are summarized in Table 2.

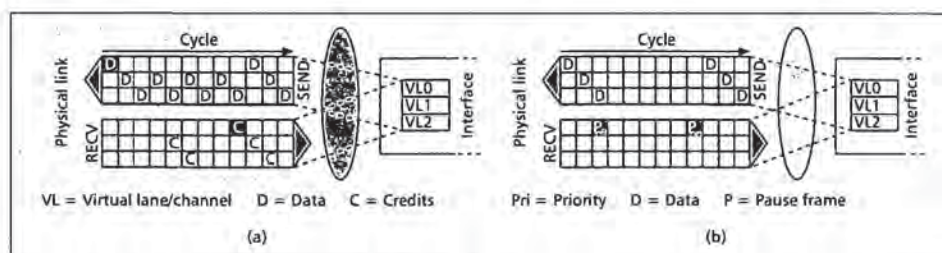
FLOW CONTROL

IBA — In order to avoid packet loss due to buffer overflows, IBA uses a point-to-point credit-based flow-control scheme. In such a scheme, the downstream side of a link keeps track of the available buffer resources (credits) by decreasing a credit counter whenever buffer space is allocated and increasing the credit counter whenever buffer space is deallocated. Similarly, the upstream node keeps track of the available credits (i.e., the number of bytes it is allowed to send) and decreases this amount whenever it sends a packet. Whenever credits arrive from the downstream node, it increases the amount of available credits. A packet is never sent downstream unless there is room for it. At regular intervals the downstream node details credit availability to the upstream node. The update interval depends on the load — high loads increase the frequency of updates, while low loads reduce the frequency.

The concept of virtual lanes allows for grouping of virtual lanes as layers, thus making it possible to build virtual networks on top of a physical topology. These virtual networks, or layers, can be used for various purposes such as efficient routing, deadlock avoidance, fault tolerance, and service differentiation.

Term	IBA	ASI	Ethernet
Virtual link	Virtual lane	Virtual channel	Priority
Service class	Service level	Traffic class	Priority
Link layer packet	Packet	Packet	Frame
Network interface	Channel adapter	Endpoint	Network interface card

■ Table 2. Terminology used by IBA, ASI, and Ethernet.



■ Figure 1. Flow control in a) IBA and ASI; b) Ethernet.

The use of flow-control ensures that packet loss is only a result of link-transmission errors and hence the available link bandwidth is used effectively as retransmissions are not necessary.

As IBA is a layered networking technology, flow-control is performed per *virtual lane* (channel). The concept of virtual lanes allows a physical link to be split into several virtual links, each with its own buffering, flow-control, and congestion management resources. Figure 1a shows an example of per virtual lane (VL) credit-based flow-control where VL0 runs out of credits after cycle 1 (depicted by a bold D) and is unable to transmit until credit arrives in cycle 9 (depicted by a bold C). As the other lanes have sufficient credit they are unaffected and are able to use the slot that VL0 would otherwise use. Transmission resumes for VL0 when credit arrives.

The concept of virtual lanes also allows for grouping of virtual lanes as layers, thus making it possible to build virtual networks on top of a physical topology. These virtual networks, or layers, can be used for various purposes such as efficient routing, deadlock avoidance, fault tolerance, and service differentiation. Virtual lanes and service differentiation are discussed further below.

ASI — ASI also uses a link-by-link credit-based flow-control mechanism to prevent packet drops as the status of a downstream ingress queue is reflected in its upstream neighbor. Credit is updated periodically and, if needed, additional replenishment is sent, as frequent transmitters will consume their allocated credit quicker than intermittent ones.

One of the limitations of a lossless network is that it can lead to a deadlock situation when a number of resources are in a circular dependency. A typical deadlock scenario is when two nodes that initiate a request/response transaction with each other are unable to complete the transaction because the response is blocked by the other request. ASI has an end-to-end appli-

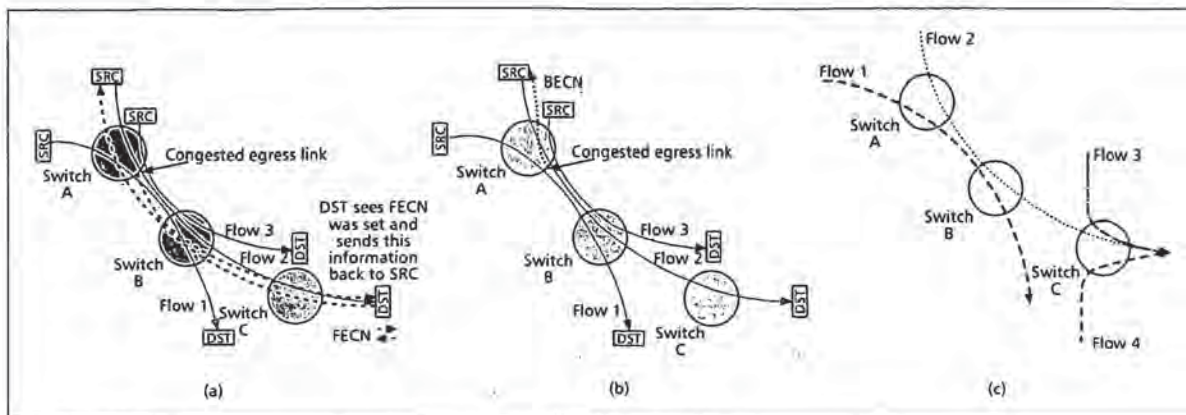
cation deadlock avoidance scheme built into its queuing structures in such a way as to break this dependency, ensuring that one of these operations can always bypass the other. Typically, certain types of PCI operations necessitates the use of such a mechanism. This deadlock avoidance is achieved by using a *bypassable* queue that maintains flow-control credit for two types of packet, *bypassable-ordered* and *bypassable-bypass*, where a bypassable-bypass is not allowed impede the progression of a bypassable-ordered packet.

During operation, when credit is abundant for both bypassable-bypass and bypassable-ordered packets, the queuing structure acts like a FIFO, but when credit reserves for the bypassable-bypass are depleted, packets in the bypassable-ordered queue with sufficient credit are allowed to pass the stalled packets until such time when bypassable-bypass credit is reallocated. These stalled packets are then given strict priority, as they have already been subjected to a delay.

It is expected that fabric management and some PCI-based operations will use the bypassable queue, while another credit-controlled unicast FIFO, the *ordered* queue, is available for other types of traffic.

Ethernet — The Ethernet link-level flow-control mechanism was specified for use within lossless networks. Flow-control is time-based, in the sense that the receiving node sends an explicit *off-message*, directing the sender to stop all transmissions for a certain amount of time so as to avoid swamping the receiver with frames that it cannot buffer. If the transmission of frames can be resumed before the time specified in the off-message has run out, an *on-message* is sent. This approach is different from the credit-based flow-control principle applied in ASI and IBA technologies, where the control-messages sent upstream reflect available buffer space in the downstream node.

In Ethernet, *xon/xoff* messages come in the form of a *pause frame* that includes a *pause-time*.



■ Figure 2. Congestion management mechanisms: a) FECN; b) BECN; c) SBFC.

The pause-time is the time that the upstream node must wait before sending the next frame. For the exchange of pause frames to work correctly, the messages must take into account the fact that there is a delay between the transmission of pause frames and their activation. This delay is a function of the propagation and processing time. It should be noted that, in order to sustain the link bandwidth, xon/xoff requires $2 \times N$ buffer space, where credit-based flow-control requires N buffer space.

The current Ethernet flow-control is limited to flow governance on a per-port basis, which undermines effective QoS provisioning when flow-control is turned on. Figure 1b shows Ethernet priorities in combination with flow-control. Since the flow-control mechanism is port-wide, the lack of buffer space for one priority affects the whole link, and any P_{off} message shuts down the link, while any P_{on} message restarts the link. We will further discuss the consequences of this below.

CONGESTION MANAGEMENT

IBA — As flow-control is a mechanism to avoid packet loss due to buffer overflows, congestion management is a mechanism to aid switches and links in the network from becoming overloaded and depleting their credit supplies. When congestion develops, a depletion of credit supplies starts and the queues begin to fill up. This process spreads upstream through the network and results in the creation and growth of congestion trees which eventually terminate at the end-nodes. Obviously, this is a bad situation for a network, as the growth of congestion trees can quickly preclude transmission of other flows (in the same virtual lane) that are not even destined towards the congested area.

IBA supports three mechanisms to control congestion. These are forward explicit congestion notification (FECN), static rate control, and a head-of-queue drain mechanism.

Forward explicit congestion notification is used to inform a packet's destination that it was subjected to congestion while traversing the network. This is achieved by setting an FECN flag in the packet's header. This flag is observed by the destination, which can signal the source about congestion either by sending a congestion notification packet or, when acknowledgments are used, by

setting the congestion flag in the next acknowledgment for the packet in question (see Fig. 2a for an illustration of FECN). In this figure, the link from switch A to switch B is oversubscribed. The FECN flag is set at this point and when the destination sees a packet that has this flag set, it sends the FECN status to the source. For the sake of clarity, we only show FECN for one flow.

A switch also has the ability to inform a source end-node of various levels of congestion through the use of up to 16 different congestion thresholds. A switch can identify itself as either the root (cause) of congestion or as a victim of congestion. If a virtual lane has exceeded a given buffer occupancy threshold and has available credits, the switch considers itself the root of congestion. Otherwise, if the virtual lane has exceeded a given threshold and is out of credits, it considers itself a victim of congestion.

Static rate control is used at the end nodes to reduce the injection rate after receiving either a congestion notification or an acknowledgment with the congestion flag set. The arrival of successive congestion notifications leads to further reductions, while a subsequent rate increase is based on a time-out that is relative to the latest congestion notification. The available static rates are selected from a congestion-control table, where each entry defines an acceptable injection rate.

IBA also supports a head-of-queue drain mechanism that ensures a switch queue is drained after a timeout. This is useful in the case where a destination has stopped consuming packets.

ASI — Congestion as a result of credit depletion is typically a transient condition, so ASI attempts to handle it using a localized congestion-control mechanism known as status-based flow-control (SBFC). The use of SBFCs allows a downstream switch to signal the congestion status of one of its egress ports to its upstream neighbor, thus allowing it to change its scheduling mechanism so that packets destined for the congested (downstream) port are given a lower priority. As a result, congestion will only have an impact on packets contributing to the congestion while packets heading for an uncongested port will remain unaffected. The use of SBFCs is illustrated in Fig. 2c, where the onset of flows 3 and 4

Even though end-to-end congestion notification is not explicitly supported within ASI, a regular ASI packet has an FECN field that can be used to inform a destination end-node that, while it was en route, it was subjected to congestion.

traversing switch C results in congestion in switch B affecting the throughput of flow 1 (that is not even destined towards the congestion). The use of SBFCs provides switch A with a one-hop look-ahead with regards to congestion, making it possible to lower scheduling priorities for packets destined for switch C.

The SBFC procedure should be invoked in advance of credit depletion so as to prevent or limit the growth of congestion trees. SBFCs are issued to order a total cessation of traffic or for limited time periods.

End-nodes also play a role in managing congestion as, ultimately, they contribute to the problem. Traffic injection into the fabric is limited by the use of connection queues with an associated injection rate. These connection queues can isolate traffic by class, destination, and chosen route. Each queue has an associated token bucket that controls the average rate of injection while also supporting a limited number of bursts.

Even though end-to-end congestion notification is not explicitly supported within ASI, a regular ASI packet has an FECN field that can be used to inform a destination end-node that, while it was en route, it was subjected to congestion. This information could be used by the destination node to signal the source node, requesting it to limit its injection rate for the given flow.

Ethernet — Currently the Ethernet standard does not offer any congestion-management mechanisms but, with the development of Backplane Ethernet, congestion-management techniques are under investigation by the IEEE 802.3ar Congestion Management Task Force. This work has not been completed as yet, but one of the goals is that congestion-notification messages are targeted directly to the end-nodes causing the congestion.

One solution under consideration is the use of a backward explicit congestion notification (BECN), which has a shorter control loop than the IBA/ASI FECN approach since there is no need to go all the way to the destination before returning a congestion notification to the source. In Fig. 2b we illustrate the use of BECN where a message is sent to the source from the point where the congestion is observed, in this case, switch A. The inclusion of a mechanism to reduce traffic injection that is based on the degree of congestion is also under consideration.

Congestion management supported at the link level ensures that Ethernet also remains agnostic with respect to higher-layer protocols, which is an important issue for interconnect protocols.

SERVICE DIFFERENTIATION

IBA — We have already seen IBA's support for virtual lanes, a key feature for supporting service differentiation. The independent resources dedicated to each virtual lane in combination with IBA's mechanisms for differentiation between lanes forms the core of IBA's QoS capabilities. Certain traffic may be assigned a low-priority best effort, for example, while others may require strict latency and jitter guarantees.

The three key mechanisms to achieve service differentiation in IBA are service level to virtual lane mapping, the weighting of virtual lanes, and their classification as either high or low priority. A

service level denotes the type of service a packet receives as it travels toward its destination, which is similar to the packet marking approach used in Differential Services as specified by IETF. Since there is not necessarily a one-to-one relation between the number of virtual lanes and service levels, a table is kept which contains a mapping from one to the other. IBA supports a maximum of 16 virtual lanes and service levels, where lane 15 is a control-lane dedicated to management traffic.

Apart from the control-lane, which has strict priority over everything else, other virtual lanes can be classified as either having high or low priority. Thus, by assigning a packet to a certain service level and setting the service level to map to a particular virtual lane, packets can be classified with either a high or low priority. High-priority traffic will preempt low-priority traffic, but in order to ensure forward progression of low-priority packets, a parameter called limit of high priority (LHP) is used. The LHP is the maximum number of packets that can be scheduled on high-priority lanes before a packet must be selected from a low-priority lane.

Arbitration between individual virtual lanes of the same priority is carried out using a weighted fair arbitration scheme. Each virtual lane is scheduled in table order and assigned a weight indicating the number of bytes it is allowed to transmit during its turn.

ASI — ASI supports differentiated services using traffic classes that isolate flows into various classes. Eight traffic classes are mapped to virtual channels, enabling flow prioritization throughout the fabric. Up to 20 virtual channels are supported: eight ordered, eight bypass, and four multicast. Fabric management messages must use the bypassable queuing structure and assigned the highest priority (TC7). These TC7 packets are given strict priority during scheduling.

ASI supports two different egress scheduling mechanisms: table-based and minimum-bandwidth. The table scheduler is a legacy PCI Express scheduler; while a minimum bandwidth scheduler allows for differentiated services for the various virtual channels while also providing a minimum share to each one to avoid starvation, usually by using a variant of weighted fair queuing. The ASI specification recommends the use of the minimum bandwidth scheduler.

During link negotiation, both processors negotiate the number of virtual channels to be employed where the lowest common number of virtual channels will be chosen and processors may adapt their queuing structures to ensure compliance.

Ethernet — In 1998 Ethernet was extended with a priority mechanism in order to support differentiated services. It is based on priority tagging of packets and an implementation of multiple queues within the switches in order to discriminate packets based on eight different levels of priority. The standard specifies Strict Priority Queuing (SPQ), but WFQ is also supported by most vendors.

There is an obvious conflict between the Ethernet priority tagging concept and the use of a port-based flow-control mechanism. Recall that the flow-control (the pause frame) is non-discriminatory. As it is port-based it will "shut down" the

link without any regard to the priority of the traffic (illustrated in Fig. 1b). This causes a situation where congestion spreads, which may result in service degradation as both low and high priority traffic will be subjected to an increase in latency and jitter. The relevant standardization bodies are responding to this problem with a suggestion that the Ethernet flow-control concept is extended to support a granularity similar to the priority mechanism (i.e., eight levels) realizing class based flow-control. This is implementable using some reserved fields in the MAC control frame.

Another consequence of Ethernet's port-based flow-control is that it cannot offer virtual layer networking for the purpose of effective deadlock-free routing and performance enhancements, as opposed to IBA and ASI [8]. The concept of virtual layer networking should not be confused with Ethernet's VLAN (Virtual Local Area Networking) feature, which is primarily a mechanism for assigning end-nodes in VLAN groups (identified by the VLAN tag) limiting the broadcast domain [9]. VLAN can also be used to implement multiple spanning trees in an Ethernet network yielding performance gains compared to single spanning tree routing, as it shuts down fewer links [10]. However, since each VLAN does not have a separate queue the union of multiple spanning trees may deadlock when flow-control is turned on.

QOS CAPABILITIES ASSESSMENT

The definition of QoS is very broad and interpreted differently dependent on the application domain, but in terms of interconnection networks, four properties are central; lossless operation, effective bandwidth distribution, minimum latency and minimum jitter.

A network should operate fairly and efficiently. Fairness is achieved by distributing the bandwidth based on service classes while efficiency equates to keeping latency and jitter low, by preventing the onset of congestion. An inescapable fact is that packets are subjected to delay while traversing a network, as packets are queued and scheduled according to various priorities.

A network becomes over-subscribed as a result of the generation of excess traffic at its edge. This traffic can result in the onset of congestion and mechanisms should be in place to limit performance degradation and prevent saturation. Typically, this over-subscription is transient but it can have a detrimental effect on performance.

We have seen some of the capabilities that the technologies have to deal with these issues, bandwidth distributed between varying classes allows for segregation of traffic as the onset of congestion in one does not affect the other. The three technologies have various mechanisms in place with regard to congestion. Common amongst these mechanisms is that they attempt to reduce delay and jitter by preventing the network from reaching a saturated state.

A loss-less network reduces the need for retransmissions and, when compared to lossy networks, we can expect increased network utilization but at a cost of congestion tree growth in the face of congestion.

Table 3 summarizes the QoS features available

Feature	IBA	ASI	Ethernet
Flow control	Credit-based	Credit-based	XOn/XOff
Congestion control	FECN	FECN, SBFC	Under review
Rate control	Static	(Semi) Dynamic	Under review
Service classes	16	8	8
Virtual channels	16	20	8
Priority scheduling	WFQ	Table, MinBW	SPQ, WFQ (optional)

■ Table 3. Quality of service features of IBA, ASI, and Ethernet.

in the three technologies. They are all capable of supporting lossless networking. IBA and ASI deploy a credit-based flow-control concept and Ethernet an XOn/XOff technique. Although the XOn/XOff scheme produces less control traffic than credit-based flow-control, it requires twice the buffer space. Apart from this the differences are negligible, and the three technologies can be considered equal with regards to lossless operation.

One side effect of flow-control is that, when a link runs out of buffers or is paused, a congestion tree will spread upstream from the point of congestion, reducing throughput and increasing latency for even the smallest flow. Both IBA and ASI support forward explicit congestion notification (FECN), which allows switches to notify end-nodes about congestion. Furthermore, IBA supports static rate control that enables end nodes to reduce injection rates based on predefined congestion thresholds and static rates.

ASI supports a parameterized token bucket algorithm for injection rate control as well as an SBFC mechanism to handle transient congestion. This gives a switch a one hop look-ahead when making its scheduling selection, allowing it to hold back packets which are likely to experience congestion downstream. In Ethernet congestion control is nonexistent, but ongoing work is needed to add this feature in the future. Currently, backward explicit congestion notification (BECN) with rate reduction, as well as end-node injection rate control, is under consideration. The BECN method improves upon FECN by introducing a shorter control loop, as the notification message goes directly from a switch to the source end node, without traveling via the destination. A dynamic injection mechanism is possible with the inclusion of information regarding the state of the observed congestion. However, until this is a reality, Ethernet has no support for congestion control.

Arguably, the most important feature needed is service differentiation, which allows two (or more) types of traffic to be treated differently. In IBA this is supported by a combination of service levels and virtual lanes, which can be categorized as high or low priority. IBA supports a total of 16 service levels that can be mapped to a maximum of 16 virtual lanes. The use of weighted fair queuing allows for a fair allocation of bandwidth, while high and low priorities make it possible to separate latency-sensitive traffic from other traffic. Furthermore, IBA supports strict

As with most technology, there is no "one glove fits all" answer to which is the best; rather, the applicable problem area should govern the choice of technology. Fundamentally, previous investment and familiarity with the technology will play a big part in this decision.

priority scheduling of high-priority traffic when it is deemed necessary, but this should be subject to admission control so as to avoid starvation.

ASI supports eight traffic classes, which can be mapped to 16 unicast virtual channels: eight ordered and eight bypassable. Each channel is supported by a weighted fair-queuing scheme in the form of the minimum bandwidth scheduler, but makes no distinction (except for fabric management messages) between high and low priority apart from the weights.

Ethernet supports up to eight priorities but its use is limited by the coarse flow-control. As described above, Ethernet's pause-based flow-control is port-based, which makes it impossible to exert flow-control without penalizing all eight priorities. This should be remedied in the future as per-priority flow-control is currently under consideration by the 802.3ar task force. But until this work is realized, Ethernet's service differentiation support is poor as compared to the offerings from IBA and ASI.

Meeting latency guarantees in lossless networks is difficult because of the effect of back-pressure. The congestion trees created by back-pressure increase latency for all packets that are part of the tree and, as the network size increases, the theoretical upper bound of latency grows exponentially [11]. A combination of service differentiation, congestion management, and efficient switch architecture contributes greatly to reducing this problem.

CONCLUSIONS

We have presented an overview of IBA, ASI, and Ethernet with a focus on their QoS features, followed by a short discussion of their strengths and weaknesses. The main features of each technology are summarized in Tables 1 and 3. It should come as no surprise that Ethernet has the weakest support for QoS, but as this is currently subject to attention, it could change in the near future. IBA and ASI have a rather similar feature set, with ASI supporting an advanced congestion management scheme, while IBA has a great deal of flexibility when it comes to configuration for service differentiation.

The success of these technologies as interconnects is dependent on many factors, and not all are related to their performance abilities. Currently, Ethernet has some QoS limitations that may impede its attempt to break into the intrasystem market, but these issues are under review. ASI has evolved from the PCI intrasystem to an intersystem interconnect, a fact which gives it a lot of strength but is no guarantee for further adoption. IBA, however, has found its niche in storage and high-performance computing, thus making it hard for other technologies without equal or superior features to make an impact. But, as with most technology, there is no "one glove fits all" answer to which is the best; rather, the applicable problem area should govern the choice of technology. Fundamentally, previous investment and familiarity with the technology will play a big part in this decision.

REFERENCES

- [1] F. J. Alfaro, J. L. Sanchez, and J. Duato, "Qos in Infiniband Subnetworks," *IEEE Trans. Parallel and Distrib. Sys.*, vol. 15, no. 9, Sept. 2004.

- [2] S.-A. Reinemo et al., "Admission Control for Diffserv-Based Quality of Service in Cut-Through Networks," T. M. Pinkston and V. K. Prasanna, Eds., *Proc. High Performance Computing — HPC 2003: 10th Int'l. Conf.*, Hyderabad, India, Dec. 2003, LNCS, Springer, vol. 2913, pp. 118–29.
- [3] InfiniBand Trade Assn., "InfiniBand Architecture Specification," 1.2 ed., Oct. 2004.
- [4] ASI Special Interest Group, "Advanced Switching Core Architecture Specification," rev. 1.1, Aug. 2004.
- [5] D. Mayhew and V. Krishnan, "PCI Express and Advanced Switching: Evolutionary Path to Building Next Generation Interconnects," *11th Symp. High Perf. Interconnects*, 2003.
- [6] IEEE 802.3-2002, "LAN/MAN CSMA/CD Access Method," 2002.
- [7] ANSI/IEEE Std 802.1D, "Media Access Control (MAC) Bridges," 1998.
- [8] O. Lysne et al., "Layered Routing in Irregular Networks," *IEEE Trans. Parallel and Distrib. Sys.*, vol. 17, no. 1, 2006, pp. 51–65.
- [9] IEEE 802.1Q-2003, "Virtual Bridge Local Area Networks," 2003.
- [10] S. Sharma, "Viking: A Multispanning-Tree Ethernet Architecture for Metropolitan Area and Cluster Networks," *INFOCOM 2004*, vol. 4, Mar. 2004, pp. 2283–94.
- [11] S.-A. Reinemo, T. Skeie, and O. Lysne, "Applying the Diffserv Model in Cut-Through Networks," *Proc. 2003 Int'l. Conf. Parallel and Distrib. Processing Techniques and Apps.*, June 2003, pp. 1089–95.

BIOGRAPHIES

SVEN-ARNE REINEMO (svenar@simula.no) received a Cand.Sci. degree in computer science from the University of Oslo in 2000. He is currently a Ph.D. student at Simula Research Laboratory and the University of Oslo, where he is participating in the ICON project. His current research focus is on quality of service in interconnection networks.

TOR SKEIE (tskeie@simula.no) is an associate professor at Simula Research Laboratory and the University of Oslo. He received an M.S. degree in computer science in 1993 and a Ph.D. degree in computer science in 1998, both from the University of Oslo. He has several years of experience as a researcher in the interconnect domain. His work mainly focuses on scalability, effective routing, fault tolerance, and quality of service in switched network topologies.

THOMAS SØDRING [M] (tsodring@simula.no) is a postdoctoral researcher at Simula Research Laboratory working on the ICON project. He received a B.S. degree in computer applications in 1998 and a Ph.D. degree in computer applications in 2002, both from Dublin City University. His primary research interests are now within the area of interconnects and the development of generic solutions for routing, fault tolerance, and quality of service. He is a member of the ACM.

OLAV LYSNE [M] (olavly@simula.no) is a research director at Simula Research Laboratory and a professor in computer science at the University of Oslo. He received an M.S. degree in 1988 and a Dr.Sci. degree in 1992, both from the University of Oslo. His early research was in the field of algebraic specification and term rewriting. In recent years he has mainly been working in the area of interconnects, focusing on effective routing, fault tolerance, and quality of service. In this field he has been a member of the program committees of several of the most renowned conferences, has participated in a series of European projects, and has published around 70 academic papers.

OLA TØRUDBAKKEN [M] (ola.torudbakken@sun.com) is a senior staff engineer for Sun Microsystems Inc., Oslo, Norway. He received an M.S. degree from the University of Oslo, Department of Informatics in 1994, and since then has been working with high-performance interconnects, switch fabrics, networking, and server systems. From 1994 to 1996 he worked as a research scientist at the Sintef Research Center, Oslo, Norway. He worked at Dolphin Interconnect Solutions from 1996 to 2000, where he held various R&D positions. He currently holds one U.S. patent, and has more than 16 U.S. patents pending. He has published several papers in the field of interconnection networks and has participated in several standardization activities.

Lecture 11:

Architectural Support for Virtual Machines

Department of Electrical Engineering
Stanford University

<http://eeclass.stanford.edu/ee282>

EE282 - Fall 2005

Lecture 10 - 1

Christos Kozyrakis

Announcements

- PA-1: due on 11/8
 - If you have not started yet, you are in trouble
- Report:
 - Why optimizations worked and why you stopped at some point
- Projects: half-way review on 11/11
 - Prepare short presentation
 - Expected to have preliminary results

EE282 - Fall 2005

Lecture 10 - 2

Christos Kozyrakis

Review of Previous Lecture:

Architectural Support for Operating Systems

- OS: a very important application for architects
- Architectural support for OS
 - Operating modes
 - Processes
 - Support for fast context switching
 - Virtual memory
 - Watch out for interactions with caching
 - Watch out for TLB performance & coverage
 - Mutual exclusion
 - Atomic memory operations

EE282 - Fall 2005

Lecture 10 - 3

Christos Kozyrakis

Today's Menu:

Architectural Support for Virtual Machines

- Definition and type of Virtual Machines
- System virtual machines
 - Motivation
 - When can you virtualize an architecture
 - CPU virtualization issues
 - Memory virtualization issues
 - IO virtualization issues
- Based on 2005 HotChips Tutorial by Jim Smith and Rich Uhlig
 - <http://www.hotchips.org/hc17/program/tutorials.htm> for more info

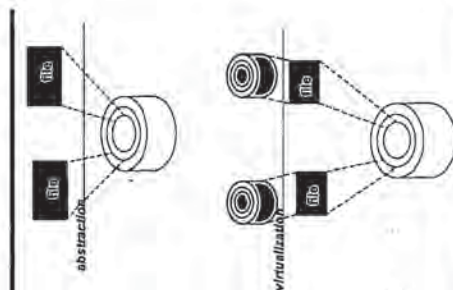
EE282 - Fall 2005

Lecture 10 - 4

Christos Kozyrakis

Virtualization

- Systems are built on levels of abstraction
 - Higher level hide details at lower levels
 - Example: files are an abstraction of a disk
- Virtualization is similar to abstraction
 - Except details not necessarily hidden
 - Example: construct Virtual Disks
 - As files on a larger disk
 - Map state
 - Implement functions
- Virtual machines (VM)
 - Do the same thing with the whole "machine"



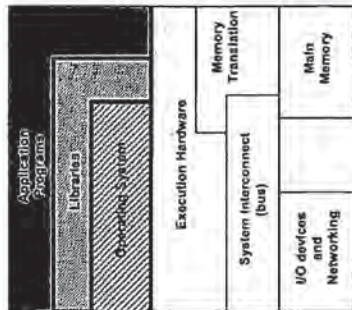
EE263 - Fall 2005

Lecture 10 - 5

Christos Kozmopoulos

What is the "Machine"?

- Many different perspectives:
 - OS developer
 - Full ISA, devices, ...
 - Compiler developer:
 - User ISA and OS calls (ABI)
 - Application programmer
 - User ISA and library calls (API)
- Leads to many types of VMs
 - **System VMs (our focus today)**
 - IBM VM/360, VMware, Xen
 - Process VMs
 - IA-32 EL, FX132, Dynamo
 - HLL VMs
 - JVM, CLR
 - Co-designed VMs
 - Crusoe and IBM Daisy



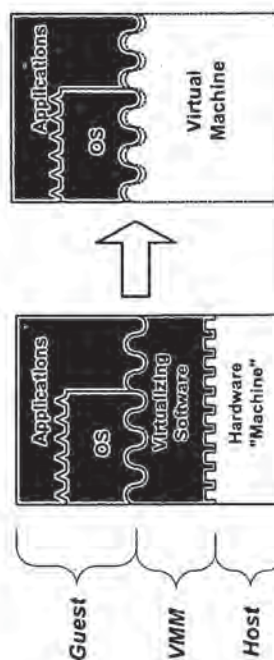
EE263 - Fall 2005

Lecture 10 - 6

Christos Kozmopoulos

Virtual Machines

- Add Virtualizing Software to a Host platform
- and support Guest process or system on a Virtual Machine (VM)
 - Example: System Virtual Machine



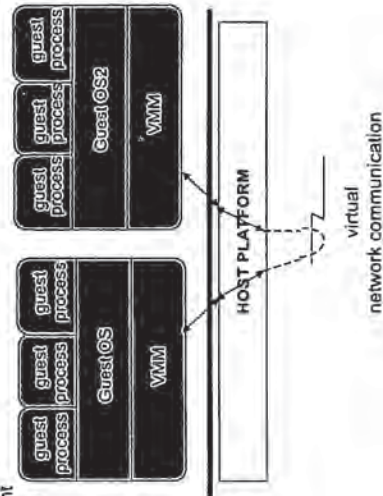
EE263 - Fall 2005

Lecture 10 - 7

Christos Kozmopoulos

System Virtual Machines

- Provide a system environment
 - Constructed at ISA level
 - Persistent
- Examples:
 - IBM VM/360
 - VMware
 - Xen



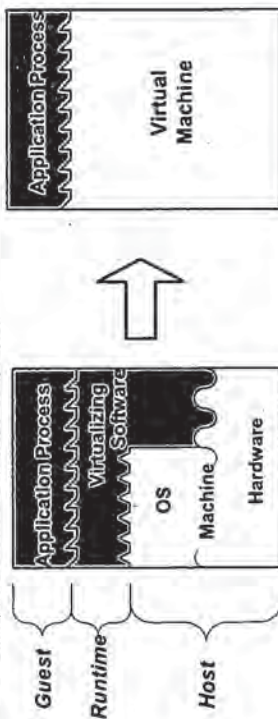
EE263 - Fall 2005

Lecture 10 - 8

Christos Kozmopoulos

Process VMs

- Execute binaries with extra optimizations or different ISA
 - Couple at ABI level via Runtime System
 - Not persistent
- Examples: IA-32 EL, FXI32, Dynamo



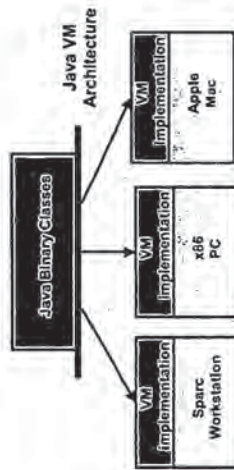
EE262 - Fall 2005

Lecture 10 - 9

Christos Kozyrakis

High-level Languages (HLL) VMs

- Java and CLI are recent examples
- Binary class files are distributed
- "ISA" is part of binary class format
- OS interaction via APIs (part of VM platform)



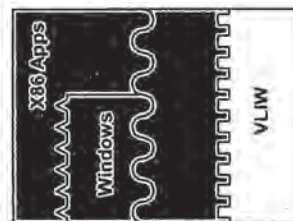
EE262 - Fall 2005

Lecture 10 - 10

Christos Kozyrakis

Co-Designed VMs

- Perform both translation and optimization
 - VM provides interface between standard ISA software and implementation ISA
 - Primary goal is performance or power efficiency
- Use proprietary implementation ISA
 - They cure to binary compatibility
- Examples:
 - IBM Daisy
 - Transmeta Crusoe
 - Apple Rosetta

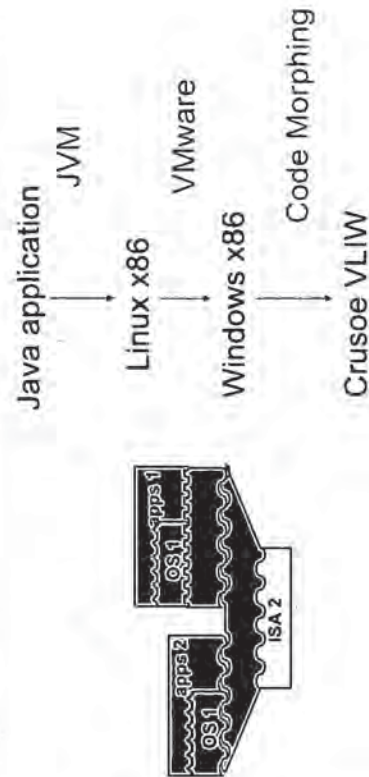


EE262 - Fall 2005

Lecture 10 - 11

Christos Kozyrakis

VM Composition

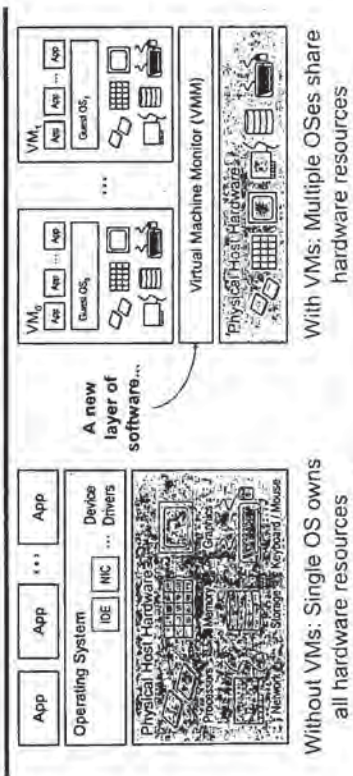


EE262 - Fall 2005

Lecture 10 - 12

Christos Kozyrakis

System Virtual Machines (VMs)



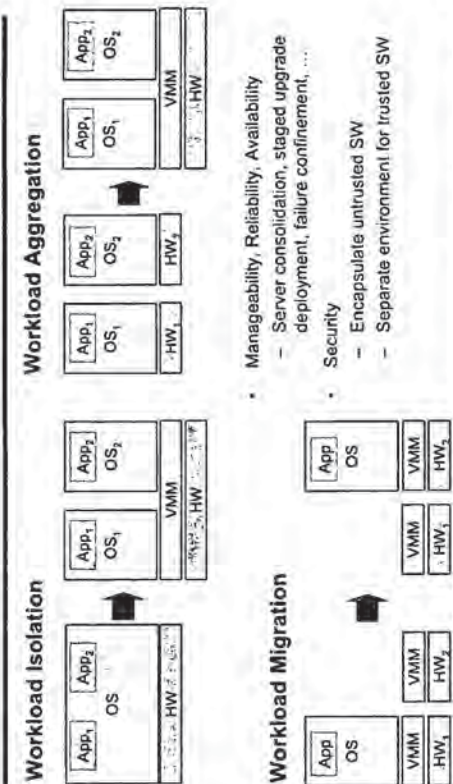
- A Virtual Machine Monitor (VMM) honors existing hardware interfaces to create virtual copies of a complete hardware system

EE262 - Fall 2005

Lecture 10 - 13

Christos Kozyrakis

Advantages of VM Model

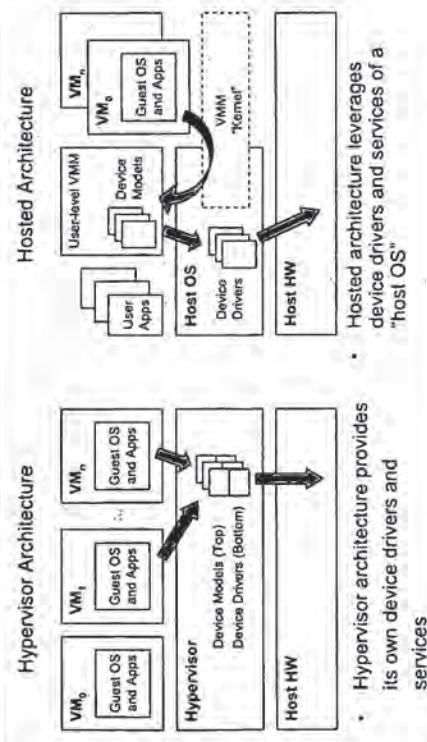


EE262 - Fall 2005

Lecture 10 - 14

Christos Kozyrakis

Some VMM Architecture Options

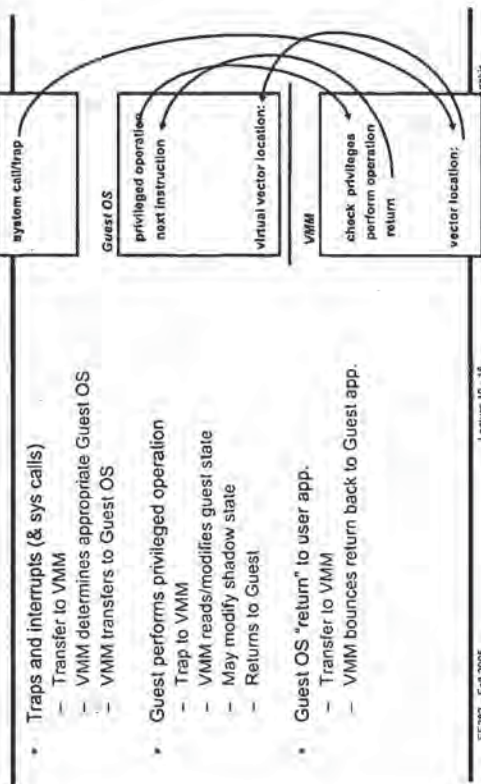


EE262 - Fall 2005

Lecture 10 - 15

Christos Kozyrakis

System Virtualization Process



EE262 - Fall 2005

Lecture 10 - 16

Christos Kozyrakis

Base VMM Requirements

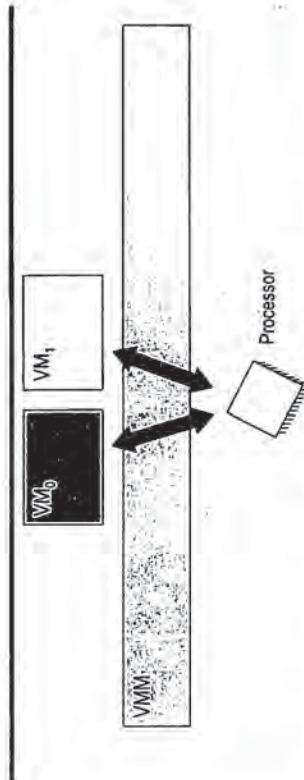
- A VMM must be able to:
 - Protect itself from guest software
 - Isolate guest software stacks (OS + Apps) from one another
 - Present a (virtual) platform interface to guest software
- To achieve this, VMM must control access to:
 - CPUs, Memory and I/O Devices
- Ways that a VMM can share resources between VMs
 - Time multiplexing
 - Resource partitioning
 - Mediating hardware interfaces

EE282 - Fall 2005

Lecture 10 - 17

Christos Kozyrakis

(1) Time Multiplexing



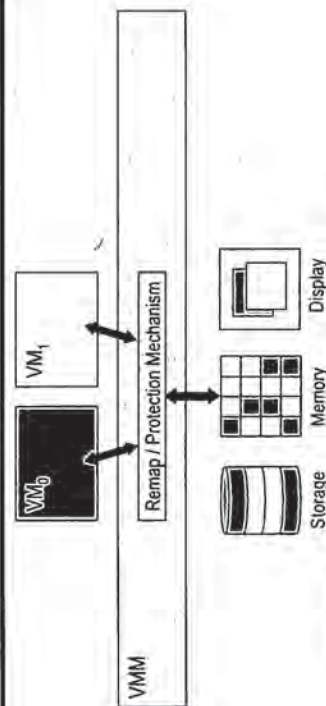
- VM is allowed direct access to resource for a period of time before being context switched to another VM (e.g., CPU resource)
 - Devil is in the details

EE282 - Fall 2005

Lecture 10 - 16

Christos Kozyrakis

(2) Resource Partitioning



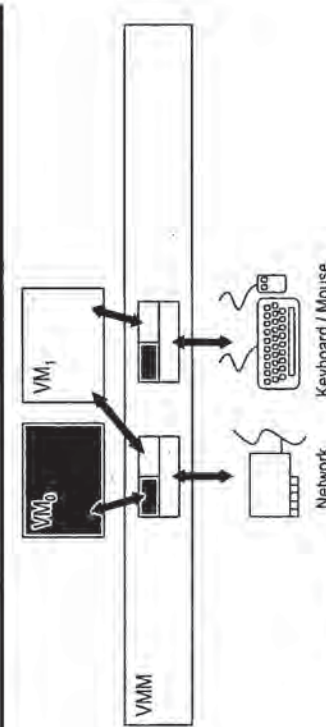
- VMM allocates "ownership" of phys resources to VMs
 - Typically involves some remapping and protection mechanism
 - Examples: physical memory, disk partitions, graphical display

EE282 - Fall 2005

Lecture 10 - 18

Christos Kozyrakis

(3) Mediating Hardware Interfaces



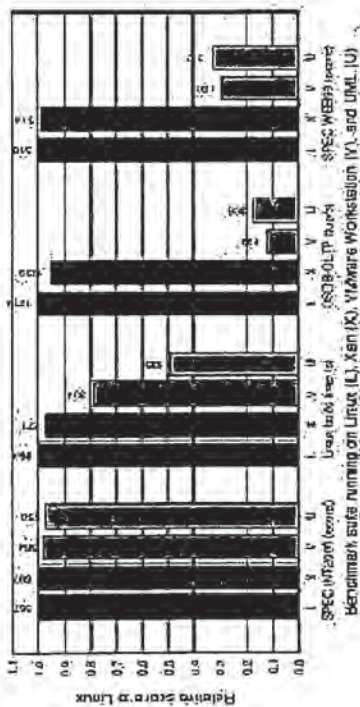
- VMM retains direct ownership of physical resource
 - VMM hosts device driver as well as a virtualized device interface
 - Virtual interface can be same as or different than physical device

EE282 - Fall 2005

Lecture 10 - 20

Christos Kozyrakis

Performance Implications of Virtualization



EE282 - Fall 2005

Lecture 10 - 21

Christos Kozymakis

When is an Architecture Virtualizable?
(Popek & Goldberg 1974)

- Basic requirement: two execution modes (user & kernel)
- Control Sensitive instructions
 - All instructions that change hardware resource allocation (or mapping)
 - Example: write TLB
- Behavior Sensitive instructions
 - All instructions whose outcome depends on hardware resource allocation
 - Example: read processor mode
- Desirable properties:
 - Efficiency
 - All non-sensitive, non kernel instructions are executed by HW directly
 - Resource control
 - No program can affect resource allocation w/o going in the VMM
 - Equivalence – same result with and without VM (excluding timing)

EE282 - Fall 2005

Lecture 10 - 22

Christina Kozynich

When is an Architecture Virtualizable?
(Popek & Goldberg 1974)

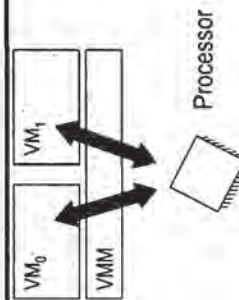
- Theorem 1:
 - Architecture is efficiently virtualizable if all sensitive instructions trap when executed in user mode
- Theorem 2:
 - Can also virtualize and ISA for which the sensitive instructions in user mode are a subset of kernel mode instructions
 - But, you must interpret instruction in guest OS
- Is your favorite ISA virtualizable?
 - Intel's x86 fails Theorem 1, but works with Theorem 2
 - Option 1: modify guest OS to work around virtualization issues
 - Xen
 - Option 2: binary translation of guest OS (modify on the fly)
 - VMware
 - Option 3: re-architect ISA
 - Intel's VT-x and AMD Pacifica

EE282 - Fall 2005

Lecture 10 - 23

Chingiz Kozymkul

CPU Virtualization: General Principles



- To virtualize a CPU, a VMM must retain control over:
 - Accesses to privileged state (control regs, debug regs, etc.)
 - Exceptions (page faults, machine-check exceptions, etc.)
 - Interrupts and interrupt masking
 - Address translation (via page tables)
 - CPU access to I/O (via I/O ports or memory-mapped I/O)

EE282 - Fall 2005

Lecture 10 - 24

Christos Kozynotis

Case Study: IA-32 CPU Virtualization

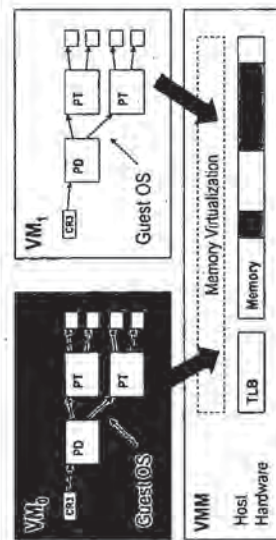
- IA-32 provides 4 privilege levels (Rings) but
 - Segmentation: distinguishes between all 4 rings
 - Paging: user mode in ring 3, supervisor mode in ring 0, 1, or 2
- Can maintain CPU control via "de-privileging"
 - VMM runs in the most privileged ring 0
 - Guest OS runs in a higher than usual ring (e.g. not 0)
- Two options
 - Guest OS in ring 0
 - Lose ring protections between guest OS / Apps
 - Guest OS in ring 1
 - Can't use paging to protect VMM from guest OS
 - VMM forced to use segment-based protections

EE282 - Fall 2005

Lecture 10 - 35

Christos Kozmopoulos

Memory Virtualization: General Principles



- Guest OS expects to control address translation
 - Allocates memory, page tables, manages TLB consistency, etc.
- But, VMM must have ultimate control over physical memory
 - Must map guest-physical address space to host-physical space

EE282 - Fall 2005

Lecture 10 - 37

Christos Kozmopoulos

Patching IA-32 for Virtualization (VT-x)

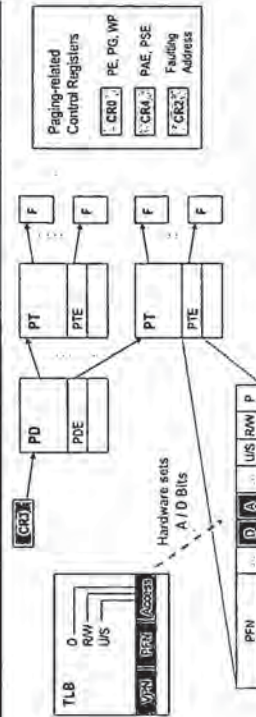
- A new operating mode enabled with VMXON/VMXOFF
 - Root Operation: Fully privileged, intended for VMM
 - Non-root Operation: Not fully privileged, intended for guest
- What got fixed
 - Guest OS still runs in ring 0
 - After VMXON all sensitive instructions can cause traps
 - VMM selects which events should trap
 - Set of control registers per "virtual CPU"
 - Describes state, behavior on entry/exit, interrupt delivery etc
 - Allows fast VMM ↔ guest OS interactions
 - Support for virtualization of virtual memory management
 - Support for virtualization of I/O
 - Interrupt delivery, marking of pages for memory-mapped IO

EE282 - Fall 2005

Lecture 10 - 28

Christos Kozmopoulos

A Case Study: IA-32 Address Translation



- IA-32 defines a hierarchical page-table structure
 - Defines linear-to-physical address translation
 - After page-table walk, page-table Entries (PTEs) are cached in a hardware TLB
- IA-32 address translation configured via control registers (CR3, etc.)
- Invalidation of PTEs signaled by OS via *INVLPG* instruction

EE282 - Fall 2005

Lecture 10 - 28

Christos Kozmopoulos

Virtualizing Page Tables: Some Options

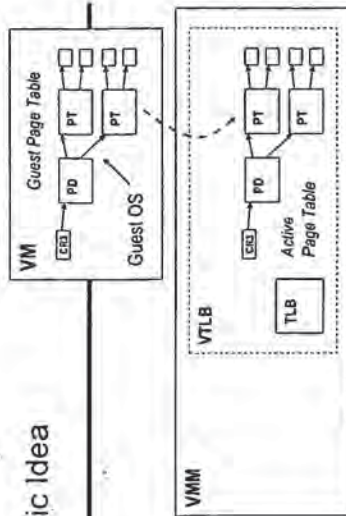
- Option 1: Protect access to guest-OS page tables (PTs)
 - Use paging protections or binary translation to detect changes
 - Upon write access, substitute remapped phys address in PTE
 - Also need VM exit on page-table reads (to report original PTE value to guest OS)
 - High overheads
- Option 2: Make a shadow copy of page tables
 - Guest OS freely changes its page tables
 - VM exit occurs whenever CR3 changes
 - VMM copies contents of *guest* page tables to *active* page tables
 - Copy operation is analogous to a TLB refill, hence: "Virtual TLB"

EE262 - Fall 2005

Lecture 10 - 28

Christos Kozyrakis

Virtual TLB: Basic Idea



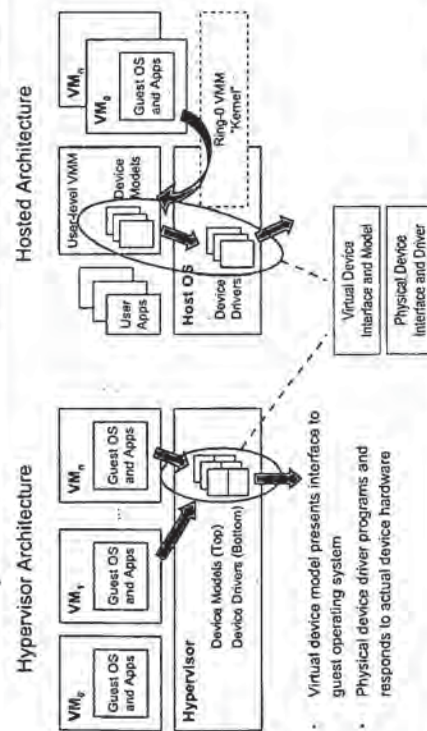
- VTLB = Processor TLB + Active Page Table
 - VMM initializes an empty VTLB and starts guest execution
 - When guest accesses memory, #PF occurs, and is sent to VMM
 - VMM copies needed translation (VTLB refill) and resumes guest

EE262 - Fall 2005

Lecture 10 - 30

Christos Kozyrakis

IO Virtualization: General Principles

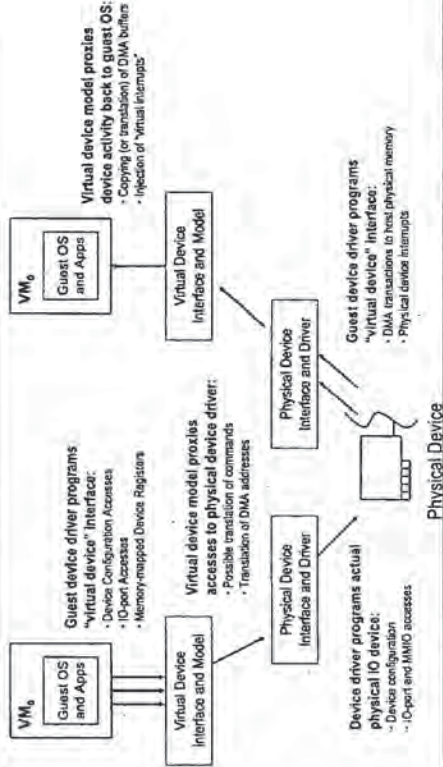


EE262 - Fall 2005

Lecture 10 - 31

Christos Kozyrakis

Virtual and Physical Device Interfaces



EE262 - Fall 2005

Lecture 10 - 32

Christos Kozyrakis

IO Virtualization Issues and Solutions

- How does one intercept accesses to memory-mapped IO?
 - Mark pages for MMIO and cause trap into VMM
- How do you interrupt guest on demands?
 - Virtualize pending interrupt register
- What if guest OS cannot be interrupted currently
 - It assumes it has interrupts off?
 - Allow guest OS to avoid interrupt until it has enabled them again
 - Expose interrupt enable mask to VMM
- Eliminate copying, interrupts, etc
 - Same techniques as with accelerating TCP/IP

EE232 - Fall 2005

Lecture 10 - 33

Christos Kozyrakis

A Big Catch

- Recursive virtualization
 - The VMM of today will be the guest OS in a few years
- Can you virtualize the resources used by the VMM?
 - Can it be done altogether?
 - Ideas?
 - Can it be done efficiently?
 - Ideas?

EE232 - Fall 2005

Lecture 10 - 34

Christos Kozyrakis

Summary of Virtual Machines

- Virtual machines: technique to virtualize a system
- Benefits
 - Security, fault-tolerance, load-balancing, ...
 - Support for legacy ISAs, OSes, etc
 - Co-development/optimization of architecture & software stack
- Architectural support for virtual machines
 - Two execution modes, no user-level sensitive instructions
 - VMM and guest OS control state tracking
 - Flexible mechanisms for virtual memory, interrupt delivery, ...

EE232 - Fall 2005

Lecture 10 - 35

Christos Kozyrakis

Packet Handling

Seminar on Transport of Multimedia Streams in Wireless Internet

Davide Astuti

Department of Computer Science
University of Helsinki
Helsinki, Finland
davide.astuti@cs.helsinki.fi

Abstract This paper describes a number of packet handling mechanisms on a network, related to classification of packets, admission control, traffic shaping, queuing and scheduling disciplines and packet discard techniques. An overview about packet handling in wireless networks is provided.

Keywords: IP QoS, FIFO, PQ, FQ, WFQ, CBQ, HTB, RED

I. INTRODUCTION

The current Internet is based on IP protocol and supports only best effort services. With the exponential growth of Internet during the last years, IP networks are expected to support not only typical services like ftp and email, but also real-time services and video streaming application. The traffic characteristics of these applications require a certain Quality of Service (QoS) from the network in terms of bandwidth and delay requirements.

The biggest problems in a network are related to the allocation of network resources, as buffers and link bandwidth, to different users. A limited amount of resources has to be shared among many different competing traffic flows in an efficient way in order to maximize the performance and the use of the network resources. The behavior of routers in terms of packet handling can be controlled to achieve different kind of services.

This paper analyzes a number of packet handling mechanisms, which are related to:

- Classification of packets
- Admission Control and Traffic Shaping
- Queuing and scheduling disciplines
- Packet discard techniques

Packet classification indicates the process of categorizing packets into flows in a router. All packets belonging to the same flow or the same class of service are ruled in a predefined manner and are processed in a corresponding way by the router. For example, all packets belonging to a certain application or related to a specific protocol may be defined to form a traffic flow. Packet classification process is needed for those services that require the distinction between different kinds of traffic.

Once packets are classified, admission control functions are performed to check whether enough network resources are available according to the packet service classification. If

resources are available, packets are handled according to their classification. Traffic shaping functions are used to control the volume of traffic entering in the networks and the rate at which packets are sent. This paper describes two methods for achieving Traffic shaping: leaky bucket and token bucket.

Afterwards, packets are scheduled into queues. Queues are managed in a way to ensure each queue gets the level of services required for its class. A number of queue disciplines are described: First-In-First-Out (FIFO), Priority Queuing (PQ), Fair Queuing (FQ), Weighted Fair Queuing (WFQ), Class Based Queueing (CBQ). A comparison between queue disciplines is provided, giving advantages and disadvantages of each discipline and the area of applicability. The Hierarchical Token Bucket (HTB) method to provide QoS in Linux is outlined.

When network congestion is experienced, packets in excess must be discarded. Packet discard cannot be avoided but can be controlled. Several techniques for handling packet discard are showed, such as Tail Drop, Early Drop, Drop Preferece and RED.

Finally, an overview of how packet handling is achieved in wireless environment is provided. When not otherwise specified, the used reference is [Huston00]. This paper is structured as follows. Packet classification is introduced in section II. Admission control and Traffic Shaping is described in section III. The queueing disciplines are described in Section IV. In Section V, packet discard methods are presented. Section VI gives an overview of packet handling in wireless environment. Finally, a summary of the paper content is provided in Section VII.

II. PACKET CLASSIFICATION

Today, Internet routers usually provide only best-effort services and packets are treated in a classical first-in-first-out manner. The need for QoS requires new mechanisms of packet handling. First of all, the router has to distinguish packets belonging to different class of services. Classes of services are specified by rules applied to incoming packets. A collection of rules is called classifier.

By inspecting the packet header, the packet is classified to several service classes and consequently is handled accordingly. The packet header determines also the congestion management of the packet. For example, a packet may be marked as low priority packet so that it may be discarded in case of congestion.

III. ADMISSION CONTROL AND TRAFFIC SHAPING

Admission control is a mechanism to estimate the level of QoS that a new traffic flow needs and to check whether enough resources are available to serve it. If network resources are available, packets are "admitted" to flow into the network.

Admission control is not mandatory. In many cases, the network accepts the traffic from a new user session without any check about the available resources. However, this behaviour may lead to an overload in the network as uncontrolled use of network services is allowed. Without admission control, the network cannot provide different service classes in an efficient way because it is not possible to prevent a service class from using more than the maximum amount of network resources reserved for that class.

A network providing differentiated services [RFC2475] must accomplish admission control. Once the packet is classified, it is passed through an admission filter to check whether enough network resources are available according to the packet service classification. The admission control monitors and enforces the use of network resources according to a certain policy. The policy criteria can be various and depends on the environment; it may include identifying users/applications or selecting traffic based on the source, destination, traffic type and so on. [RFC2753] describes a policy-based control framework for admission control.

An easy technique to provide admission control is to service flows based on arrival time. This is called CAC (capacity-based admission control). With this method, new flows are admitted until capacity is exhausted (First Come First Admitted) without any additional check. New flows are not admitted until network resources are available.

Another method to achieve admission control consists of allowing some flows to preempt others according to certain priority settings. This method has the advantage of providing an efficient premium service as the amount of resources used by the low priority traffic is controlled and limited. On the other hand, adding high priority traffic can affect the performance of lower priority traffic, so priority queued systems must use sophisticated admission control algorithms. [RFC2751] and [RFC2815] describe admission control in a priority-queued system.

Once the traffic is admitted to the network (through admission control functions), traffic shaping practices are used to control the volume of traffic entering in the networks and the rate at which packets are transmitted. In this way, the flow of packets is smoothed according to the configured profile of traffic load on the network.

Traffic shaping can be implemented using a leaky bucket filter. The bucket is essentially a regulated queue. A traffic source can produce packets at quite unsteady and unconstant rate, which trickles into the bucket. Packets flowing into the bucket are paced out in steady way at a specified rate. We can imagine that the bucket has an imaginary hole in the bottom and packets can trickle into it, in analogy with a real bucket full of water with a hole. Water trickles out of the bucket little by little, independently of how quickly the bucket is filled up. Using a leaky bucket filter does not allow sending burst of

packets, but only at a specified rate. If there is no traffic for a certain period of time, the amount of unused bandwidth cannot be used for later packets; this is achieved by using a token bucket filter.

A. Token Bucket

A finer way to implement traffic shaping uses a token bucket filter, which allows bursts of packets to be sent every now and then. The token bucket filter is a way to implement both admission control and traffic shaping. The philosophy is the same than the leaky bucket's one, however the token bucket allows having permission to send bursts of packets. Figure 1 shows the token bucket scheme. The algorithm is that every certain period of time – which determines the rate packets are served – a token is saved (tokens are marked with "T" in the figure). Each token can be associated to one packet (blue packets in the figure) or, in general, to a defined quantity of data. When a packet is sent, one of these tokens is removed. If there are no packets to send, tokens can be retained and used to send new packets later on. In the figure, four tokens are retained in the bucket; hence, a sudden burst of four packets can be served and allowed to go through the network.

When a packets arrives and there are not enough tokens retained in the bucket, the action can be various according to how the Token bucket filter is configured: the packet can be buffered and not discarded until new tokens are available; the packet can be marked and, hence, treated in particular way; the packet can be discarded.

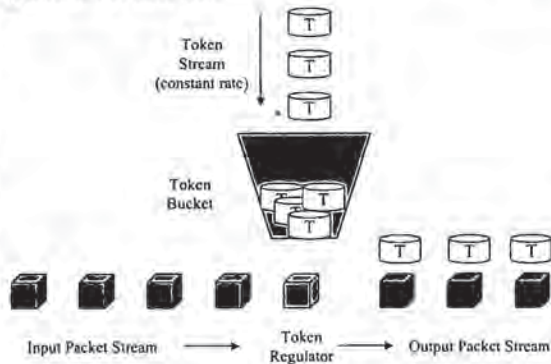


Figure 1. Token Bucket

We can see why the Token Bucket filter implements both Admission control and Traffic Shaping functions. If packets arrive at a rate exceeding the maximum allowed rate for their service class, only the number of packets matching the number of retained tokens is "admitted" to flow into the network. At the same time, the rate at which packets are sent is "shaped" as tokens are accumulated at a specified rate.

IV. QUEUING AND SCHEDULING DISCIPLINES

In a network, packets are accumulated and queued into memory buffers of routers and switches. The most common way to arrange packets is called first-in-first-out (FIFO), but various methods may be used to prioritize packets or ensure

that all packets are handled in fair manner, rather than allowing one source to use more than its share of resources.

Packets may arrive at routers in bursts from multiple devices, and a node may sometimes receive more packets than it can process. Buffers hold packets until the router can handle them. If the router is overloaded, buffers fill up and overflow. The most common method applied in this case is called tail drop and consists of dropping the new incoming packets that does not fit in the buffer. This is not the best possible way for handling queues, but it is fast.

This paragraph analyzes the operation, benefits and drawbacks for a number of queue discipline mechanisms. The analyzed queue disciplines schemes include:

- First-in-first-out (FIFO)
- Priority Queuing (PQ)
- Fair Queuing (FQ)
- Weighted Fair Queuing (WFQ)
- Class Based Queuing (CBQ)

A.FIFO

The FIFO discipline is the basic first-in, first-out queuing method. It is very simple: the first packet in the queue is the first packet that is served. In FIFO, all packets are treated in the same way: they are placed in a single queue and are served in the same order they were placed. When the queue becomes full, congestion occurs and the new incoming packets are dropped. Figure 2 shows an example of FIFO. Packets from different flows arrive at the router, which multiplexes them into the same FIFO queue in the same arrival order (1,2,3,4,5,6).

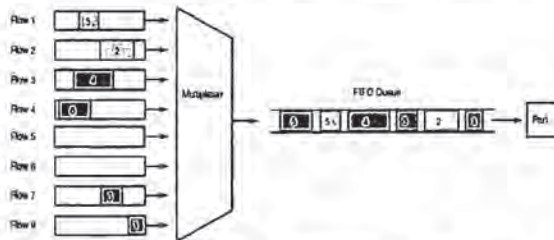


Figure 2. First-In-First-Out (FIFO) queuing [Semeria].

The main benefits of FIFO are described as follows. The FIFO technique is extremely simple when compared with other more elaborated queue disciplines. Consequently, it is a good solution for software-based routers. FIFO does not make packet reordering and the maximum queue delay can be easily calculated: it is determined by taking into account the maximum length of the queue. As long as there is no congestion and the FIFO queue is not full, the contention resolution for network resources is provided in a fast way because of the simplicity of this technique.

On the other hand, FIFO has drawbacks. Since all packets are buffered in the same queue, FIFO does not allow handling packets of different classes in different way. Therefore, it is not

possible to provide different Classes of Service. The queuing delay increases as congestion increases and such additional delay affects all queued packets. Consequently, real-time applications or, in general, applications with strict delay and jitter constraints, can suffer and experience bad performance.

Moreover, during network congestion FIFO benefits non-congestion-responsive flows (UDP) over congestion-responsive flows (TCP). When a TCP packet is lost, TCP infers that the network is congested and reduces its sending rate. On the other hand, when a UDP packet is lost, UDP does not respond to such an event and continue the transmission at its normal sending rate. Consequently, FIFO queuing can result in a form of bandwidth starvation for the congestion-responsive flow (TCP) [FF97]. The most severe example of this situation is where an UDP flow transmits packets at a sending rate higher than the available bandwidth. Under these load conditions, the FIFO queue will be always full, apart one empty slot at the tail of the queue every now and then. Such a slot is the unique one a TCP flow can use since it adapts and reduces its sending rate according the estimated network congestion [FF97]. FIFO does not help to resolve this problem of fair allocation of network resources.

B.PQ

Priority Queuing (PQ) is a method to provide in a simple way a differentiated service. This technique uses multiple queues. Queues are serviced with different levels of priority and queues with higher priority are serviced first. Packets are placed in one of the queues according to their classification. Packets are scheduled from a certain queue only if the higher priority queues are empty. Packets are scheduled in FIFO order within each priority queue. In case of congestion, packets are dropped from lower priority queues.

Figure 3 shows an example of Priority Queuing. In this case, three priority queues are used. Flow 3 is classified as highest priority flow, flows 2 and 7 as middle priority flows and the other flows as lowest priority flows. Firstly, packets from the highest priority queue are served in FIFO order. Only once the highest priority queue is empty, the remaining queues will be served using the same principle.

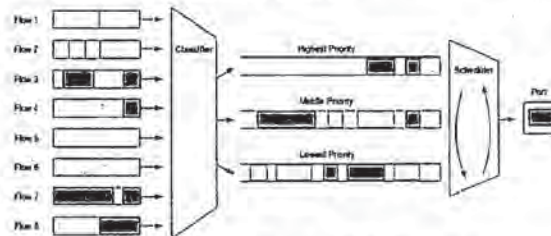


Figure 3. Priority Queuing (PQ) [Semeria].

The main benefit of PQ is that packets of different classes can be managed using different queues and, consequently, a certain class of traffic can be handled in different way than another one. Another advantage is that the PQ technique is relatively simple when compared with other more elaborated queue disciplines.

The drawbacks of PQ are outlined as follows. If the amount of highest priority traffic is excessive then the lower priority queue may not get any service until the highest priority traffic is served completely. During this while, the queues allocated to lower priority traffic may overflow. As a result, the lower priority traffic may experience a large delay or, in the worst case, a complete resource starvation. A solution to this important problem consists of limiting the maximum amount of highest priority traffic (in general, limiting all the priority queues) that can be accepted using appropriate admission control policies.

The FIFO drawback regarding the interaction between TCP and UDP flows (bandwidth starvation for the TCP flows) is not resolved using PQ. A simple use of PQ to cope with such a problem would be to map TCP flows into a higher priority queue than UDP flows in order to compensate the non-congestion-responsive behavior of UDP. This is not a good solution because the TCP algorithm will try to use all the available bandwidth causing a bandwidth starvation for the lower priority queues.

C.FQ

Fair Queuing (FQ) has been designed to avoid the problem of PQ discipline, where one flow can monopolize and use all the available bandwidth of the network and, hence, lead the lower priority flows to bandwidth starvation. In FQ, arriving packets are classified into different flows and stored into a queue dedicated to that particular flow. The round-robin algorithm is used to service all the queues, so that queues are served in a fair way and one source cannot use more than its share of network bandwidth. Figure 4 illustrates an example of FQ. Each flow is redirected to its own queue. Queues are scheduled in round-robin way.

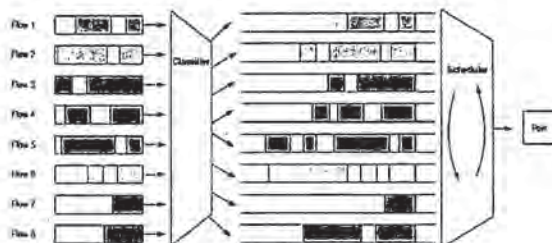


Figure 4. Fair Queuing (FQ) [Semeria].

The main advantage of FQ is that bursty flows do not affect the overall performance. If a flow is too bursty, then only its queue will be filled up, without any interference with any other flow.

The drawbacks of FQ are outlined as follows. The scheduler takes a packet at a time from each queue regardless of the packet length. For example, if a queue contains bigger packets than other queues, then that queue will use a larger portion of the total network bandwidth and it will take more time to be served. Furthermore, the FQ algorithm is more complicated than FIFO and PQ; hence a software

implementation (the most common) may result in performance problems. In general, FQ is good to share the same portion of bandwidth among many flows, but it cannot be used for handling different flows bandwidth requirements. Moreover, there is no easy way to provide real-time services.

A variant of FQ is Stochastic Fair Queuing (SFQ) [McKe90]. The difference between FQ and SFQ is in the way the arriving flows are mapped into queues. The number of queues is limited and a hash function is used to separate traffic flows into separate FIFO queues, which are served in a round-robin way. The hash function takes into account the source/destination address, the port number and other protocol parameters. If the number of incoming flows is larger than the number of queues, multiple flows will get hashed to the same queue. It may happen that the hash function maps many flows in the same queue meanwhile there are queues without any data or few data to be sent. In order to avoid such an unfairness situation, the hashing algorithm is perturbed at predefined time intervals so that the mapping between flows and queues periodically changes. Perturbation may however cause packet reordering to occur. SFQ is one of the disciplines available in Linux Traffic Control [LARTC].

D.WFQ

The idea of the Weighted Fair Queuing (WFQ) algorithm can be easily explained as a combination of PQ and FQ algorithms. As in the FQ method, all queues are served so that there is no bandwidth starvation, but some queues have more weight in a sense that they receive more service. In other words, a weight is given to each queue to assign different priorities to the queues. Packets are stored into the appropriate queue according to their classification.

The above discussion gives a general idea of the algorithm; the complete algorithm is a little bit more complicated and it is an approximation of the Generalized Processor Sharing (GPS) system [Keshav97]. The algorithm is explained as follows. A finish time is assigned to each packet taking into account the link bandwidth, the number of queues, the weight of queues and the packets length. Afterwards, the scheduler serves the queue where the packet with the minimum finish time is stored. The finish time is used to build the order in which the packet will be transmitted through the link.

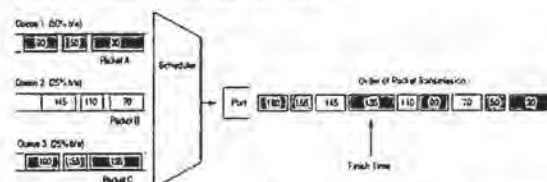


Figure 5. Weighted Fair Queuing (WFQ) [Semeria].

Figure 5 shows an example of the WFQ algorithm. The assigned finish times are only for a qualitative use. At the beginning, three packets are given to the scheduler. The queue 1 has a weight of 50%, queue 2 and 3 have a weight of 25%. Hence, the finish time for one packet in queue 1 is less than for one packet of the same length in the other queues with lower weight. In this example, the first two packets in queue 1 get

finish time of 30 and 50 respectively; therefore they are forwarded before the packet in queue 2, labeled with a finish time of 70. As Figure 5 shows, giving a high weight to a queue allows the scheduler to take more than one packet from that queue. The order of packets transmission is shown on the right part of the figure.

The main benefit of WFQ is that it provides a minimum level of network resources to each configured service class. On the other hand, WFQ is a complex algorithm requiring a per-service class state and iterative checking of all states for each packet arrival and departure. Consequently, the WFQ presents scalability problems and cannot be used in environments with high volume of traffic requiring many classes of service. One possible application of WFQ is where the number of service classes is limited to a small number, so that the computational load is bounded.

Since WFQ was invented, many improvements have been proposed, which include Class-based WFQ, SCFQ, WF2Q. The interested reader can refer to [Semeria] for more information.

E.CBQ

The main idea behind Class-Based Queuing (CBQ) is scheduling packets in the queues and guaranteeing a certain transmission rate. If there are no packets in a queue, its bandwidth is made available to other queues. The strength of this method is that it allows to cope among flows with considerably different bandwidth requirements. This is done by assigning a specific percentage of the link bandwidth to each queue. CBQ also avoids the bandwidth starvation problem of the PQ method as at least one packet is served from each queue during each service round. CBQ and WFQ may look similar as they try to achieve the benefits of PQ and FQ. One difference is that CBQ scheduling is packet-oriented and does not take into account the packet length.

The algorithm is outlined as follows. Firstly, packets are classified into a service class according certain criteria (addresses, protocol, users, etc...) and stored in the corresponding queue. Each queue is served in round robin order. A different amount of bandwidth can be assigned for each queue in two different ways: allowing a queue to send more than one packet for each service round or allowing a queue to send only one packet for each service round but the same queue is visited more than one time during the same service round.

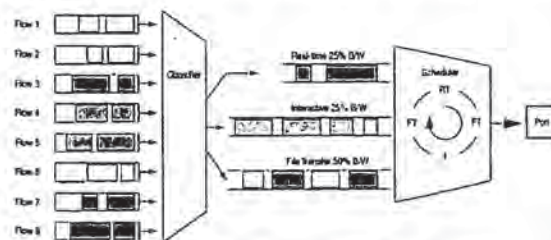


Figure 6. Class-based queuing (CBQ) [Semeria].

Figure 6 shows an example of CBQ. Incoming traffic is classified in three service classes: Real-Time (RT), Interactive (I) and File Transfer (FT). In this example, half of the available bandwidth is reserved for the file transfer service class; the remaining bandwidth is equally shared between real-time and interactive class. Using these weights (25%,25%,50%), the File Transfer queue is visited two times during one service round.

One of the main benefit of CBQ is clearly shown in this example, where we can see why the bandwidth starvation problem does not rise using the CBQ method: each service class receives a certain amount of service for each service round and the case present in PQ, where low-priority queues do not receive any service if higher priority queues contain packets, never happens.

CBQ gives control over the amount of bandwidth each service class can use and it can be easily implemented in hardware, hence there are no the performance limitations of other queue disciplines.

The main drawback of CBQ is that it can provide fair allocation of the bandwidth only if packets of all queues have the same (or comparable) size. If one service class contains longer packets than the other ones do, the service class with bigger packet size will take more bandwidth than the configured value. The interested reader can found a complete research on CBQ in [CBQ].

F.A Comparison between queue disciplines

In this paragraph, a comparison between the outlined queue disciplines is introduced as each method presents benefits and drawbacks. In addition, suitable areas of applications for each queue method are reported. Table 1 summarizes advantages and disadvantages for each queue discipline.

Queue discipline	Advantages	Disadvantages
FIFO	Extremely simple. No packet reordering. Maximum delay easy to predict.	No service class supported. Queueing delay increases as congestion increases. No fairness (bandwidth starvation).
PQ	Supports different service classes. Relatively simple.	Bandwidth starvation problem. No fairness.
FQ	Support different service classes. Bursty flows do not affect other flows. Allows share the same bandwidth among many flows.	Packets must have the same size for efficient scheduling. No way to provide real time services. No way to serve flows with different bandwidth requirements. High complexity.
WFQ	Gives a minimum amount of resources to each service class. Bursty flows do not affect other flows. Allows sharing the same bandwidth among many flows.	High complexity.
CBQ	Gives a minimum amount of resources to each service class. Limited complexity.	Packets must have the same size for fair scheduling.

Table 1. Pro and cons of queuing disciplines

FIFO discipline is very simple and therefore can be easily scaled in large queues and used in fast-switching networks. FIFO behavior is easy to predict and no packet reordering is performed. However, FIFO is not fair among different flows, hence bursty traffic sources may monopolize the link bandwidth and there is no way to handle different traffic priorities.

Even though FIFO has important drawbacks, it can be used both in networks without QoS and network with QoS enabled. In the former case, FIFO is worth to be used as it is very efficient and fast. In the latter case, FIFO can be used in those QoS-enabled networks where the traffic admission control functions are performed at the edge of the network and the core network has enough resources to handle the admitted traffic in the edge. The main requirement is that admission control functions must be able to efficiently limit the amount of traffic admitted in the network so that it can flow into the core network. This kind of configuration is beneficial for very large networks. In this way, services requiring high computational power are performed at the edge of the network where traffic load is generally low and the core network can use FIFO queues since its high-speed requirements.

PQ allows handling different service classes and offers a low-jitter, low-loss service to the highest-priority traffic. On the other hand, if the amount of highest priority traffic is excessive then flows classified with a lower priority may not get any service until the highest priority traffic is served completely. Consequently, PQ can be applicable in those environments where a single high-priority traffic source (or a small number compared with the total traffic) is present. A proper use of admission control functions can limit the maximum amount of highest priority traffic; hence, PQ can be used in those environments where Admission Control is configured to compensate the drawbacks of the PQ discipline.

The main advantage of FQ is that bursty flows do not affect the performance of other flows. FQ tries to share a fair portion of bandwidth among many flows, but it cannot serve flows with different bandwidth requirements. The typical use of FQ is at the edge of the network, where users connect to their service provider. Implementations of FQ typically classifies packets into 256, 512 or 1024 queues using a hash function that is calculated across source/destination addresses, UDP/TCP port numbers and IP ToS byte [Semeria]. At the edge of the network, a user usually generates a limited number of flows; using FQ, each flow or a small number of flows can be assigned to each queue, providing isolation between flows and reducing the impact a bursty flow can have on the other flows.

WFQ tries to share the available bandwidth among all the traffic flows in fair way, by taking into account the weight each queue has. In this way, bursty traffic sources do not affect performance of other flows. However, the complexity of WFQ is high and grows up in proportion to the number of active flows. This is a strong limitation for applying WFQ in high-speed networks. On the other hand, WFQ can be effectively used in medium-low speed access networks.

CBQ has no the drawback of PQ, where low-priority queues do not receive any service if packets are in higher priority queues, and gives to every service class a certain service level. Since the number of queues in CBQ is statically configured, the service granularity that CBQ can provide is limited to the number of configured queues. Network environments where packet sizes are very variable are not suitable for CBQ, as CBQ does not provide fair allocation of bandwidth when packets with a variable length have to be handled.

Both CBQ and WFQ are meant to overcome the limitations of PQ and FQ algorithms. In this sense, WFQ offer the best performance, however its high complexity limits the range of applications it can be used. Both WFQ and CBQ give a minimum amount of resources to each service class, but CBQ needs packets of the same length to provide fair scheduling. On the other hand, CBQ is simpler than WFQ.

Finally, we can state that there is not a queuing mechanism suitable for all environments. The choice depends on the specific characteristics of the network environment. In order to provide efficient packet service, queuing disciplines cannot be used alone but in conjunction with a proper set of packet classification functions, admission control, and packet discard techniques.

G.HTB in Linux

Hierarchical Token Bucket (HTB) is a method to have bandwidth distribution among several service classes in a very flexible way. HTB is actually the most used technique in Linux Traffic Control [HTB] to provide QoS. HTB is composed of an arbitrary number of token buckets filters arranged in hierarchical way. We explain how HTB works using an example. Let us suppose we want to share the link among several service classes. Figure 7 shows a simple example where five service classes shares the same link, each one using a maximum amount of the total link bandwidth. The mail service class shares only 0% of the total bandwidth in case of congestion, which means that mail is serviced only if there is bandwidth available from other services. The CBQ queue discipline may be used to achieve this kind of structure.

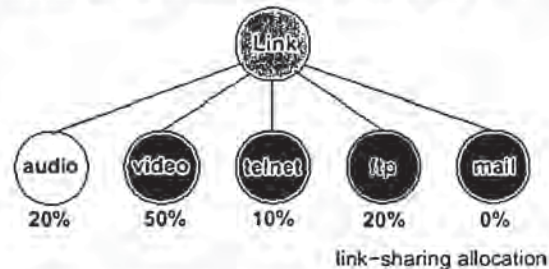


Figure 7. An example of link sharing between service classes.

HTB allows a more dynamic structure of link sharing. Not only it allows sharing the bandwidth among service classes, but permits link sharing between organizations, protocol families, and individual connections inside a service class. Figure 8 shows an example of hierarchical link sharing structure. In this example, we share the same link among three agencies, using respectively 50%, 40% and 10% of the total bandwidth. Inside the service class "Agency A", we can create a hierarchy of other service classes like real-time, telnet and ftp. Each "sub service class" shares the bandwidth portion defined at the higher layer (real time uses 30%, telnet and ftp uses 10% each one). If the telnet class for Agency A has nothing to send, its assigned portion will be used from real-time and ftp classes on the same hierarchy level. At higher level, if Agency A does not use all its assigned portion of the total bandwidth, it will be distributed among the other classes.

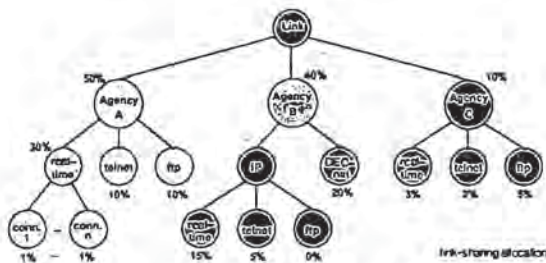


Figure 8. An example of hierarchical link sharing structure (HTB).

As we can see, the HTB structure is much more flexible than the "flat" one showed in Figure 7 because we have multiple levels of bandwidth distributions, not only one. One drawback of HTB is its complexity. Performance limitations may arise when a complex hierarchical link sharing structure is used.

V. PACKET DISCARD TECHNIQUES

When network congestion is experienced, the number of arriving packets at a router overwhelms the available bandwidth of the link and the router queues try to cope with such a situation. As queues are finite memory buffers, they will fill up and, consequently, packet discard is necessary. Packet discard events can be controlled by choosing appropriate size of queues. However, huge queue sizes may cause excessive packet delay, as the current packet needs to wait all its predecessors before being served.

Packet discard should not be used as a response for a congestion situation, as any discarded packet has already used a certain amount of network resources up to the point it is discarded. However, packet discard cannot be completely avoided as queues may always overflow. Packets should be discarded in fair way, which means that not only the same source gets its packets discarded but the discarded packets belong to all the present traffic sources. In this paragraph, different packet discard methods are described, each one with advantages and disadvantages, taking a look at the fairness they provide. The described methods include Tail Drop, Early Drop, Drop Preference and RED.

A. Tail Drop

Tail drop is the most simple and common packet discard algorithm and it is commonly used with the FIFO queue discipline. Once the queue gets full, the new incoming packets are dropped. Packets continue to be dropped until new slots are available in the queue.

The main drawback of the Tail Drop algorithm is its low fairness. A sudden traffic burst from one source may fill up all the available slots of the queue and the new incoming traffic would be discarded. A solution to this problem is placing many service classes within the same queue. Each service class has a maximum amount of slots in the queue. When such a limit is overcome, packets of that service class are discarded. In this way, there is still space for other service classes even if one of

them exceeded its available space. The drawback of this solution is that packet discarding happens even if there is available free memory for accepting new packets in the queue.

B. Early Drop

Early Drop is a technique meant to give a form of fair resource sharing among different flows. The main idea is to discard a priori bursty flows before the queue overflows, so that there is still space for other flows. According to such a definition, placing many service classes in the same queue (see the previous paragraph about Tail Drop) is one instance of the Early Drop method.

A way of implementing Early Drop consists of monitoring each flow and its traffic characteristics. When the queue length grows up to a certain limit, packets belonging to flows arriving at high rate or bursty flows can be discarded. This is a precise way to achieving Early Drop. However, it may present scalability problems in a loaded system with thousands of distinguished flows. In such an environment, selecting packets to be discarded in random way is usually preferred as in a congested environment the packet that will be chosen would probably belong to one of the flows with the highest bandwidth requirements.

TCP is one of the protocols that respond to packet drops as it supposes that packet drops are a clear signal of network congestion [TCP]. This is why Early Drop can also be used to signal the flow source that congestion is happening and, indirectly, reducing its sending rate.

C. Drop Preference

Another method for choosing which packet to drop consists of using external information for discarding the packet. When congestion arises and the queue length reaches a predefined limit, discarded packets are not the ones in the tail of the queue, but packets marked with a "drop preference" flag (or low priority flag). Such a flag can be set by the source application or by the admission control functions.

In this way, when congestion happens, the first flows to be discarded are the "less important" flows, whereas the most critical traffic is preserved and forwarded. In IP networks, the Drop Preference mechanism is associated with admission control functions. For example, all the packets belonging to a certain service class exceeding the maximum allowed bandwidth for that class are marked so that they are forwarded if bandwidth is available; otherwise they are the first packets to be discarded in case of congestion.

D. Random Early Drop (RED)

Random Early Drop [RED] is a method for discarding packets to cope with the network congestion problem. The philosophy of RED consists of assist and help congestion-responsive protocols, like TCP, to prevent and avoid network congestion. TCP [TCP] assumes that a packet drop is clear indication of congestion and reduces accordingly its sending rate, going back into slow start or reducing its congestion window size. With other packet discard methods, packets are dropped when congestion is already in act (buffers are full); the idea of RED is to start packet discarding before the queues are

full so that congestion is prevented and TCP adjusts its sending rate before congestion happens.

RED uses statistical methods to drop packets before the router queue overflows. It keeps track of the average queue size and discards packets when the average queue size overcomes a certain threshold. In more detail, the average queue size is calculated every time a new packet arrives and is compared to two thresholds: a *minimum* and a *maximum* threshold. When the average queue size is less than the minimum threshold, nothing happens. When the average queue size is greater than the maximum threshold, the arriving packet is dropped. If the average queue size is between the minimum and the maximum threshold, the arriving packet is dropped with a certain probability p . The RED algorithm is summarized as follows:

```

for each packet arrival
    calculate the average queue size  $avg$ 
    if  $min\_thresh \leq avg < max\_thresh$ 
        drop the arriving packet with probability  $p$ 
    else if  $max\_thresh \leq avg$ 
        drop the arriving packet
  
```

The average queue size is calculated using an exponential weighted moving average [FJ93], meaning that if the queue has recently been empty for the most of the time, RED will not react to a sudden burst of packets as if a major congestion event were happened. However, if the average queue length remains near to the maximum threshold, RED will assume congestion and start dropping packets at a higher rate. [FF93] shows a complete analysis of RED behaviour when bursty traffic arrives.

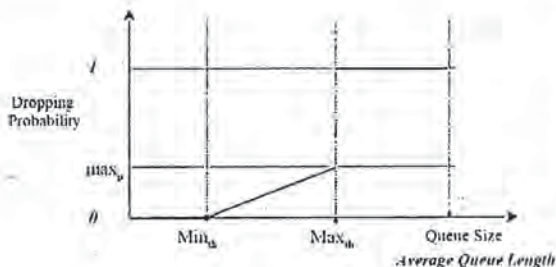


Figure 9. Dropping probability vs Average Queue Length

Figure 9 shows the dropping probability p in function of the average queue length avg . As we can see, p is zero while the average queue length is less than the minimum threshold and it grows up to a max_p value as the average queue length increases. The max_p parameter may be set up to one, depending on the configuration.

Many research studies [RED] shows that RED is an effective mechanism for providing congestion avoidance at routers in cooperation with TCP. Moreover, RED discards

packets in more fair way when compared to other packet discard techniques.

A variant of RED consists of not dropping packets but simply marking them when the average queue size overcomes the minimum threshold. In this case, RED is used in conjunction with Explicit Congestion Notification (ECN) [RFC2481]. The router sets a congestion notification bit in the packet and forwards the packet to the receiver. At this point, the receiver can inform the sender to slow down its sending rate.

One critical issue of RED consists of tuning its parameters, like the minimum and maximum threshold and the weight to calculate the average queue size. It is difficult to find a set of parameters that offers good performance in all kinds of traffic profiles. Depending on the traffic source behaviour and the set of chosen parameters, RED may start to drop packets too early or too late, nullifying the reasons of why it has been developed.

A lot of variants have been proposed to improve RED, like Adaptive version of RED (ARED) and Weighted RED. ARED tries to set the RED parameters dynamically in order to better respond to different congestion situations. WRED is a technique of dropping packets based on the type of traffic, on the destination or other criteria. The interested reader can find a complete survey of RED and its variants in [RED].

VI. PACKET HANDLING IN WIRELESS ENVIRONMENT

The characteristics of mobile wireless networks make providing and implementing QoS and, in general, packet handling in an efficient manner more challenging. Wireless links present higher packet loss rate than wired links, variable and higher latency. Moreover, when a mobile node moves from one cell to another one, resources must be allocated in the new point of attachment to provide a continuous service to the user. The lack of resources in the new point of attachment, the time needed to renegotiate the session characteristics and to change the routing path may lead to a fluctuation in the QoS level provided to the user.

In a wireless environment, the IP layer and the link layer should interact and cooperate in order to provide an efficient packet handling as the knowledge of the amount of available radio resources and in general the current status of the wireless medium are needed to provide QoS.

In most cases, the interface between the network layer and the data link layer is a simple FIFO queue with a limited amount of space to temporarily store packets until they are transmitted over the link. In order to cope with the characteristics of wireless links (higher loss rate, higher and variable latency) and to provide QoS, the link layer may use different queues for scheduling the packets received from the IP layer. For example, packets requiring a reliable service may be stored in a special queue at the data link layer so that error-recovery mechanisms (FEC, ARQ) are achieved to recover packet errors.

In order to provide various QoS levels, the data link layer may use multiple queues and a scheduler, in analogy with the IP layer. The same queue and scheduling disciplines outlined in paragraph IV may be used at the data link layer. Using multiple

queues at the link layer may be necessary to provide a better service. However, this is a layer function violation as packet scheduling should be done at the IP layer.

Figure 10 shows four different approaches for providing QoS with the assistance of the link layer [MKLLSR01]. The network-controlled approach (Figure 10a) is the most common model in which the interface between the network layer and the link layer is a simple FIFO queue. Packet classification, queuing and scheduling are performed at the IP layer. This model is suitable with low-latency links, but presents the head-of-line blocking problem. Such a problem is exacerbated with a high-latency link. For example, if one packet is lost and the link layer retransmits it, all other packets must wait (behind the head of the line) until the retransmission reaches the receiver or the sender gives up and discards the packet. With high-latency links, this may cause a significant delay for delivering the remaining packets.

Figure 10b shows the link-controlled approach, where the network layer has only one queue and the data link layer incorporates a packet classifier, multiple queues and a scheduler. This model does not present the head-of-line blocking problem. On the other hand, the link layer should perform functions, such as packet classification, belonging to the IP layer and thus leading to a significant function layer violations. Moreover, if IP layer security [RFC2401] is used then the link layer packet classifier may not work properly as the transport/application headers of IP packets may be encrypted and thus unreadable at the link layer [MKLLSR01].

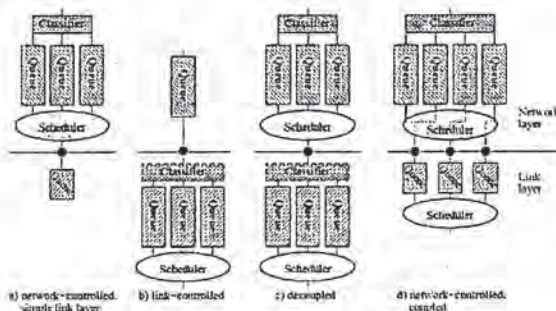


Figure 10. Four scheduling models [MKLLSR01]

Figure 10c shows the decoupled approach, which combines the approaches (a) and (b) and performs packet classification, queuing and scheduling both at the IP and data link layer. This model does not present the head-of-line blocking problem, but requires an accurate configuration in order to avoid bad scheduling interactions between IP and link layer. For instance, a high priority packet forwarded first by the IP layer may be delayed and transmitted later on after other packets by the link layer. In other words, the order of packets transmission over the link may change due to the interactions between the IP and the link layer schedulers. Moreover, the duplication of all the packet handling functions makes the system inefficient.

Figure 10d shows the network-controlled, coupled approach, which means that the IP layer classifies, schedules

and forwards each packet to the correct link layer queue. This is done by defining a special interface between the IP and the link layer – called convergence layer – which allows the IP layer to have the information about the status of link layer queues and to control the link layer queuing and scheduling. In this way, the IP layer can store the scheduled packets into the appropriate link layer queue according to their class of service; it can calculate how long a packet can wait in a particular data link queue and decide what to do with those packets exceeding the maximum tolerable delay. Using such an approach, the harmful scheduling interactions between IP and data link layer outlined previously for the case (c) are avoided and the link layer scheduling can handle packets in a more efficient way as it is controlled by the IP layer.

VII. SUMMARY

This paper describes a number of basic methods for packet handling. The main steps for providing different levels of QoS are described: Packet Classification, Admission Control, Traffic Shaping, Queue disciplines, Packet Discard. Several queue disciplines are analyzed, like FIFO, PQ, FQ, WFQ and CBQ. Advantages and disadvantages for each method are outlined and they are summarized in Table 1. The HTB discipline used in Linux Traffic Control is described. Packet Discard methods like Tail Drop, Early Drop, Drop Preference and RED are analyzed. Finally, methods for handling packets in wireless environment are described.

After such a survey, we can conclude that there is not a unique set of packet classification, admission control, traffic shaping, queuing disciplines and packet discard methods to handle any kind of network environment. In order to give an efficient packet service, the use of all these techniques must be harmonized and configured so that the specific characteristics of the used network environment, the traffic workload and the needed level of QoS are taken into account.

REFERENCES

- [CBQ] <http://www.icir.org/floyd/cbq.html>. References on CBQ (Class-Based Queuing).
- [FF97] S. Floyd, k. Fall, "Router Mechanisms to Support End-to-End Congestion Control", 1997.
- [FJ93] S. Floyd, V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance", August 1993.
- [HTB] Martin Devera, "HTB Linux queuing discipline manual", <http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm>.
- [Huston00] Huston, Geoff, "Internet Performance: Survival Guide: QoS strategies for Multiservice networks". John Wiley & Sons, February 2000.
- [Keshav97] S. Keshav, "An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network", Addison-Wesley, 1997.
- [LARTC] Linux Advanced Routing & Traffic Control, <http://lartc.org/>.
- [McKe90] P. McKenney "Stochastic Fairness Queuing", IEEE INFOCOMM'90 Proceedings, San Francisco, 1990.

[MKLLSR01] J. Manner, M. Kojo, A. Laukkanen, M. Liljeberg, T. Suihko, K. Raatikainen, "Exploitation of Link Layer QoS Mechanisms in IP QoS Architectures", ITCOM 2001, Denver, Colorado, 20 - 24 August, 2001.

[RED] <http://www.icir.org/floyd/red.html>. References on RED Queue Management.

[RFC2401] S. Kent, R. Atkinson, "Security Architecture for the Internet Protocol", IETF, November 1998.

[RFC2475] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, W. Weiss, "An Architecture for Differentiated Services", December 1998.

[RFC2481] K. Ramakrishnan, S. Floyd, "A Proposal to add Explicit Congestion Notification (ECN) to IP", January 1999.

[RFC2753] R. Yavatkar, D. Pendarakis, R. Guerin, "A Framework for Policy-based Admission Control", January 2000.

[Semeria] Chuck Semeria, "Supporting Differentiated Service Classes: Queue Scheduling Disciplines", Juniper Networks.

[TCP] Richard Stevens, "The protocols (TCP/IP illustrated, Volume 1)", Addison Wesley.

Using OpenFabrics InfiniBand for HPC Clusters

The OpenFabrics Alliance was formed to resolve issues with hardware and software interoperability and to deliver open source software for Remote Direct Memory Access fabric technologies like the InfiniBand architecture. Major initiatives by this alliance have resulted in the creation of an open source Linux® OS-based software stack for InfiniBand, which is rapidly gaining acceptance in the high-performance computing field.

BY MUNIRA HUSSAIN, RINKU GUPTA, AND TONG LU

Related Categories:

Dell PowerEdge servers

High-performance computing (HPC)

InfiniBand

Interconnects

OpenFabrics

Visit www.dell.com/powersolutions for the complete category index.

Effective server resource utilization requires a balance among CPU speed, memory bandwidth, and I/O performance. However, traditional bus-based architecture is limited in its ability to meet fast-growing CPU performance. To eliminate the I/O bus architecture bottleneck, the InfiniBand Trade Association (IBTA), consisting of leading computer technology companies like Dell, Intel, Sun, and AMD, was formed to develop specifications for the InfiniBand architecture. By the end of 2000, the IBTA had defined InfiniBand technology, which is a switched fabric I/O architecture that replaces the traditional server PCI bus with a high-speed, low-latency serial I/O interconnect to deliver a point-to-point, scalable interconnect infrastructure. In contrast to proprietary interconnects, InfiniBand is based on industry standards, enabling it to evolve and scale.

InfiniBand has shown itself to be a compelling high-performance interconnect technology for performance-focused

cluster environments. Because of its performance and scalability,¹ InfiniBand has been adopted in high-performance computing (HPC) cluster environments for communication-intensive applications, including those in the fields of aerospace and defense, oil and gas exploration, fluid dynamics, weather modeling, and biology and life sciences. According to the June 2006 TOP500 Supercomputer Sites list of the world's most powerful computers, InfiniBand is rapidly gaining acceptance as a high-performing cluster interconnect, and the number of InfiniBand-based supercomputers has grown 33 percent since November 2005.²

In addition to its benefits for Interprocess Communication (IPC), InfiniBand has been successfully recognized and utilized in the storage systems field. By delivering a converged communication and storage interconnect, InfiniBand can help simplify data center management and help reduce total cost of ownership. The adoption of

¹ For more information, see "Exploring InfiniBand as an HPC Cluster Interconnect," by Onur Celebioglu, Ramesh Rajagopalan, and Rowen Ali, *Dell Power Solutions*, October 2004, www.dell.com/downloads/global/power/ps404-20040140-Celebioglu.pdf; and "Using PCI Express Technology in High-Performance Computing Clusters," by Rinku Gupta, Saeed Iqbal, Ph.D., and Andrew Bachler, *Dell Power Solutions*, November 2005, www.dell.com/downloads/global/power/ps405-20050226-Gupta.pdf.

² November 2005 and June 2006 "Interconnect" lists, TOP500.org, www.top500.org/stats/26/conn and www.top500.org/stats/27/conn.

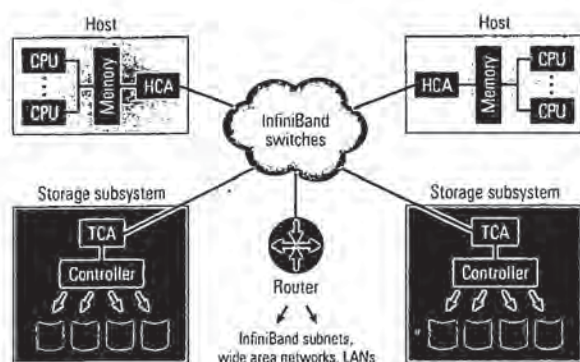


Figure 1. InfiniBand hardware architecture

InfiniBand in the server and storage market, in turn, is driving the demand for development of a complete software stack to support InfiniBand products from different vendors. To fulfill this demand, the OpenFabrics Alliance (formerly the OpenIB Alliance) was formed in June 2004 to create a standard open source InfiniBand software stack. The availability of a standard open source software stack can help promote wide adoption and interoperability across heterogeneous computing architectures.

InfiniBand hardware architecture

Established as an industry standard, the InfiniBand architecture develops a different approach toward I/O efficiency by replacing the traditional shared bus architecture with a switched fabric.³ As Figure 1 shows, the unified InfiniBand I/O fabric includes three basic components: host channel adapter (HCA), target channel adapter (TCA), and InfiniBand switches. HCAs are connection interfaces to CPUs; TCAs are links to storage, Fibre Channel networks, and other I/O nodes. InfiniBand switches provide the switched connections between HCAs and TCAs. InfiniBand routers can be used to connect multiple switched subnets.

The InfiniBand protocol is an OS bypass protocol; it provides direct access to the InfiniBand HCA and can reduce the number of user-kernel context switches and memory copies. It off-loads data movement from the server CPUs to the InfiniBand HCA, enabling communication between devices and hosts without the traditional system resource overhead associated with network protocols. With Remote Direct Memory Access (RDMA) enabled, InfiniBand adopts a zero-copy approach to transferring data between the sender's memory and the receiver's memory without involving the host CPUs. RDMA has the potential to be useful in communication-intensive applications.

InfiniBand offers traffic management through virtual lanes (VLs), creating multiple virtual links within a single physical link.

This mechanism allows a pair of linked devices to isolate communication interference from other connected devices. Each InfiniBand link can accommodate 2 to 16 VLs, which includes 1 VL for traffic management and others for packet transmission.

InfiniBand delivers high bandwidth by utilizing full duplex bidirectional links between devices. It is designed to support three data transfer rates: 1X, 4X, and 12X, with a base data rate (1X) of 2.5 Gbps, thus theoretically providing full duplex single data rate (SDR) bandwidths of 5 Gbps, 20 Gbps, and 60 Gbps. Ongoing development of InfiniBand technology with double data rate (DDR) and quad data rate (QDR) operations may help enable significantly increased throughput.

OpenFabrics software stack

The InfiniBand architecture is a complex and hierarchical structure comprising many components, libraries, programming interfaces, and modules along with multiple protocols that can be used based on application requirements. The InfiniBand interconnect gained popularity in the open source community as research and scientific computing labs adopted it. As a result, multiple versions of the software stack became available. To standardize the InfiniBand software stack, the OpenFabrics Alliance (then named the OpenIB Alliance) was formed to create a standardized Linux OS-based InfiniBand software stack that would be hardware and vendor independent. Key members of the alliance include Dell, Intel, Cisco Systems, and AMD; InfiniBand-focused companies like Mellanox Technologies, SilverStorm Technologies, and Voltaire; and research and scientific computing labs such as Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratory.

The maturity of the InfiniBand stack and the efforts of this alliance have resulted in the adoption of InfiniBand into the upstream Linux kernel. As a result, core components of InfiniBand have been accepted in the Linux 2.6 kernels (2.6.11 and later). Since its inception, the OpenFabrics Alliance has broadened its charter to include other RDMA-capable data center fabrics. The OpenFabrics Alliance now supports iWARP (RDMA on Ethernet) along with InfiniBand.

InfiniBand software architecture

The InfiniBand software architecture (see Figure 2) consists of both kernel-level and user-level components, which complement each other to help provide an end-to-end solution. In the software stack, the low-level InfiniBand kernel driver module is hardware specific and ported on top of the associated hardware. The rest of the architecture is hardware agnostic and is divided into kernel-space software and user-space software.

³ Additional details about the InfiniBand architecture can be found in the InfiniBand specification at www.infinibandta.org/specs.

Kernel-space software

The kernel-space software consists of the core InfiniBand modules and upper-layer protocols.

Core InfiniBand modules. The core InfiniBand module layer consists of the following:

- **InfiniBand verbs:** These are defined semantically in the InfiniBand specification and describe the action or function to take place. The actual translation of the semantics of a verb to the form of an application programming interface (API) is carried out on top of the hardware driver in the kernel-level software.
- **Management Datagram (MAD) services and agents:** These define an entry point for the upper layers to interact with the HCA driver and hardware.

The InfiniBand specification defines the Subnet Manager, which is responsible for topology discovery and management. It also defines a group of managers called General Services Managers, which are responsible for management and operation of the InfiniBand hardware; examples of such managers include the Performance Manager, which is responsible for retrieving performance statistics from devices, and the Communication Manager, which is responsible for managing connections between devices. Each General Services Manager typically has a corresponding agent on the InfiniBand device (for example, the Performance Manager Agent or Device Manager Agent) and uses MAD packets to request information or operations.

The MAD services layer in the kernel includes components like the Subnet Management Interface (SMI) and General Services Interface (GSI). The SMI is a unique queue pair (QP, the InfiniBand mechanism to send and receive data), called QP0, associated with InfiniBand ports to send and receive information and operation request-response packets between the Subnet Manager and its agents. Similarly, the GSI is a unique QP1 associated with InfiniBand

ports to communicate with the various General Services Managers and their agents.

Upper-layer protocols. The rest of the kernel-space software consists of upper-layer protocols like IP Over InfiniBand (IPoIB), Socket Direct Protocol (SDP), SCSI RDMA Protocol (SRP), and Internet SCSI Extensions for RDMA (iSER) in the kernel space.

User-space software

The user-space software consists of access to certain user-level InfiniBand services. InfiniBand allows user-level access to the InfiniBand hardware by providing user-space APIs. The user-level verbs API (semantically similar to the kernel-space verbs) allows upper-level middleware like Message Passing Interface (MPI) software and other applications to bypass the kernel and OS to obtain direct access to the InfiniBand hardware.

OpenFabrics Enterprise Distribution

The OpenFabrics Enterprise Distribution (OFED) is an effort by the OpenFabrics Enterprise working group—a group composed of select hardware vendors providing products based on the OpenFabrics stack—to create a commercial-quality, solution-level OpenFabrics software distribution for end users. It enables end users to deploy InfiniBand devices from various hardware vendors by using a generic software stack. The OFED distribution is a superset of the OpenFabrics software, and contains a snapshot of the OpenFabrics software and components such as MPI software (which falls outside the scope of OpenFabrics) along with the latest kernel modules taken from the OpenFabrics repository.

As of June 2006, the first release of the OFED distribution includes a comprehensive list of kernel modules and software packages such as IPoIB, SDP, SRP, User Direct Access Programming Library (uDAPL), MPI implementations like Open MPI and MVAPICH, subnet managers like OpenSM, and various performance and diagnostic tools.

InfiniBand interoperability and manageability

The OpenFabrics software stack enables hardware and software interoperability and manageability between heterogeneous InfiniBand clusters. Such capabilities have helped this software stack to contribute immensely toward the adoption of InfiniBand. For more information, visit www.openfabrics.com.

Munira Hussain is a systems engineer adviser in the High-Performance Computing Group at Dell.

Rinku Gupta is a systems engineer and consultant in the High-Performance Computing Group at Dell.

Tong Liu is a systems engineer in the High-Performance Computing Group at Dell.

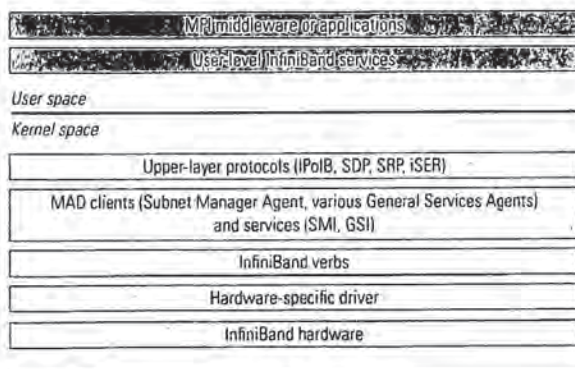


Figure 2. InfiniBand software architecture and component stack

Network Working Group
Request for Comments: 2697
Category: Informational

J. Heinanen
Telia Finland
R. Guerin
University of Pennsylvania
September 1999

A Single Rate Three Color Marker

Status of this Memo

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (1999). All Rights Reserved.

Abstract

This document defines a Single Rate Three Color Marker (srTCM), which can be used as component in a Diffserv traffic conditioner [RFC2475, RFC2474]. The srTCM meters a traffic stream and marks its packets according to three traffic parameters, Committed Information Rate (CIR), Committed Burst Size (CBS), and Excess Burst Size (EBS), to be either green, yellow, or red. A packet is marked green if it doesn't exceed the CBS, yellow if it does exceed the CBS, but not the EBS, and red otherwise.

1. Introduction

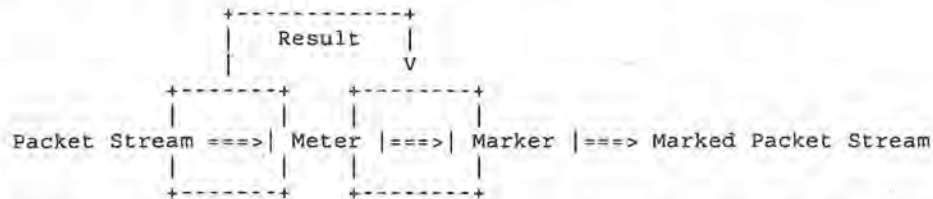
The Single Rate Three Color Marker (srTCM) meters an IP packet stream and marks its packets either green, yellow, or red. Marking is based on a Committed Information Rate (CIR) and two associated burst sizes, a Committed Burst Size (CBS) and an Excess Burst Size (EBS). A packet is marked green if it doesn't exceed the CBS, yellow if it does exceed the CBS, but not the EBS, and red otherwise. The srTCM is useful, for example, for ingress policing of a service, where only the length, not the peak rate, of the burst determines service eligibility.

Heinanen & Guerin

Informational

[Page 1]

The Meter meters each packet and passes the packet and the metering result to the Marker:



The Meter operates in one of two modes. In the Color-Blind mode, the Meter assumes that the packet stream is uncolored. In the Color-Aware mode the Meter assumes that some preceding entity has pre-colored the incoming packet stream so that each packet is either green, yellow, or red. The details of the pre-coloring process, including handling of error scenarios, and how the Meter determines the color of a pre-colored packet are DS domain specific and outside the scope of this document.

The Marker (re)colors an IP packet according to the results of the Meter. The color is coded in the DS field [RFC2474] of the packet in a PHB specific manner (see section 4 for an example).

A companion document [RFC2698] describes another three color marker, called a Two Rate Three Color Marker (trTCM), where packets are marked based on two rates and two burst sizes.

2. Configuration

The srTCM is configured by setting its mode and by assigning values to three traffic parameters: a Committed Information Rate (CIR), a Committed Burst Size (CBS), and an Excess Burst Size (EBS).

The CIR is measured in bytes of IP packets per second, i.e., it includes the IP header, but not link specific headers.

The CBS and the EBS are measured in bytes. The CBS and EBS must be configured so that at least one of them is larger than 0. It is recommended that when the value of the CBS or the EBS is larger than 0, it is larger than or equal to the size of the largest possible IP packet in the stream.

3. Metering

The behavior of the Meter is specified in terms of its mode and two token buckets, C and E, which both share the common rate CIR. The maximum size of the token bucket C is CBS and the maximum size of the token bucket E is EBS.

The token buckets C and E are initially (at time 0) full, i.e., the token count $T_c(0) = CBS$ and the token count $T_e(0) = EBS$. Thereafter, the token counts T_c and T_e are updated CIR times per second as follows:

- o If T_c is less than CBS, T_c is incremented by one, else
- o if T_e is less than EBS, T_e is incremented by one, else
- o neither T_c nor T_e is incremented.

When a packet of size B bytes arrives at time t, the following happens if the srTCM is configured to operate in the Color-Blind mode:

- o If $T_c(t) - B \geq 0$, the packet is green and T_c is decremented by B down to the minimum value of 0, else
- o if $T_e(t) - B \geq 0$, the packets is yellow and T_e is decremented by B down to the minimum value of 0, else
- o the packet is red and neither T_c nor T_e is decremented.

When a packet of size B bytes arrives at time t, the following happens if the srTCM is configured to operate in the Color-Aware mode:

- o If the packet has been precolored as green and $T_c(t) - B \geq 0$, the packet is green and T_c is decremented by B down to the minimum value of 0, else
- o If the packet has been precolored as green or yellow and if $T_e(t) - B \geq 0$, the packets is yellow and T_e is decremented by B down to the minimum value of 0, else
- o the packet is red and neither T_c nor T_e is decremented.

Note that according to the above rules, marking of a packet with a given color requires that there be enough tokens of that color to accommodate the entire packet. Other marking policies are clearly possible. The above policy was chosen in order guarantee a

deterministic behavior where the volume of green packets is never smaller than what has been determined by the CIR and CBS, i.e., tokens of a given color are always spent on packets of that color.

The actual implementation of a Meter doesn't need to be modeled according to the above formal specification.

4. Marking

The Marker reflects the metering result by setting the DS field of the packet to a particular codepoint. In case of the AF PHB [RFC2597], the color can be coded as the drop precedence of the packet.

5. Service Example

The srTCM can be used to mark a packet stream in a service, where different, decreasing levels of assurances (either absolute or relative) are given to packets which are green, yellow, or red. For example, a service may discard all red packets, because they exceeded both the committed and excess burst sizes, forward yellow packets as best effort, and forward green packets with a low drop probability.

6. Security Considerations

The srTCM has no known security concerns.

7. References

- [RFC2698] Heinanen, J. and R. Guerin, "A Two Rate Three Color Marker", RFC 2698, September 1999.
- [RFC2597] Heinanen, J., Baker, F., Weiss, W. and J. Wroclawski, "Assured Forwarding PHB Group", RFC 2597, June 1999.
- [RFC2474] Nichols, K., Blake, S., Baker, F. and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, December 1998.
- [RFC2475] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z. and W. Weiss, "An Architecture for Differentiated Services", RFC 2475, December 1998.

8. Authors' Addresses

Juha Heinanen
Telia Finland, Inc.
Myyrmaentie 2
01600 Vantaa, Finland

EMail: jh@telia.fi

Roch Guerin
University of Pennsylvania
Department of Electrical Engineering, Rm 376 GRW
200 South 33rd Street
Philadelphia, PA 19104

EMail: guerin@ee.upenn.edu

9. Full Copyright Statement

Copyright (C) The Internet Society (1999). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.



Reconstructing I/O

Keir Fraser, Steven Hand, Rolf Neugebauer,
Ian Pratt, Andrew Warfield, Mark Williamson

August 2004

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500
<http://www.cl.cam.ac.uk/>

© 2004 Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt,
Andrew Warfield, Mark Williamson

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

<http://www.cl.cam.ac.uk/TechReports/>

ISSN 1476-2986

Reconstructing I/O

Abstract

We present a next-generation architecture that addresses problems of dependability, maintainability, and manageability of I/O devices and their software drivers on the PC platform. Our architecture resolves both hardware and software issues, exploiting emerging hardware features to improve device safety. Our high-performance implementation, based on the Xen virtual machine monitor, provides an immediate transition opportunity for today's systems.

1 Introduction

Device drivers are one of the most troublesome aspects of commodity operating systems — a weakness that has received scant attention as the PC platform has evolved, driven by the overriding goal of affordable performance. It is now no longer sufficient to merely provide ever-improving data speeds; as these systems are increasingly used in business-critical applications, it is essential to review their management and dependability.

Driver code is written for, and runs within a specific OS. For purposes of execution, the two are inseparable on modern systems. This entanglement leads directly to three problems that urgently need addressing given the growing stake PCs hold in today's enterprise server environments:

1. **Dependability:** The lack of isolation between execution of driver and OS code sacrifices dependability. Driver errors often cause catastrophic system crashes.
2. **Maintainability:** Device drivers must be rewritten for each OS, and driver code is difficult and expensive to develop and maintain.
3. **Manageability:** The troubleshooting and resolution of driver-related problems is often difficult and time consuming in the ongoing administration of a system.

These concerns reflect the roots of the PC architecture as a desktop platform. Although inconvenient, system crashes and tedious diagnosis of hardware problems are accepted as

part of the PC experience by workstation users and administrators. However, in recent years the PC has supplanted mainframes and special-purpose operating systems in the enterprise. When every second of downtime impacts revenue, the traditional limitations of the architecture can no longer be tolerated.

This paper presents a set of changes intended to transition toward dependable, maintainable and manageable systems. We present a new system architecture which addresses fundamentally unsafe issues in the current I/O model, such as unrestricted device DMA. Our architecture is built upon three key ideas. First, we mitigate the risks of device misbehaviour by enforcing isolation between device-granularity protection domains. Second, we introduce a set of simple, unified interfaces between OS and driver software. These interfaces provide the required separation of concerns between these hitherto conflated aspects of systems software. Finally, we unify the control and configuration of devices in a single OS-agnostic system interface.

Our approach is not merely an academic proposal. We have taken advantage of our experience with high performance virtualization techniques to construct a complete implementation of this proposal on existing hardware. Our prototype solves all the above problems, except where constrained by the shortfalls of existing hardware, and provides a transition path toward a new system architecture as well as immediate benefit to present-day systems. In the following subsections we discuss each of the fundamental concerns more specifically.

1.1 Dependability

Drivers on the PC architecture are frequently blamed as a leading cause of system instability [1]. The fact that driver code runs with the same privilege and in the same address space as the operating system means that even simple pointer errors may compromise system stability. However, driver misbehaviour is not confined to just pointer errors:

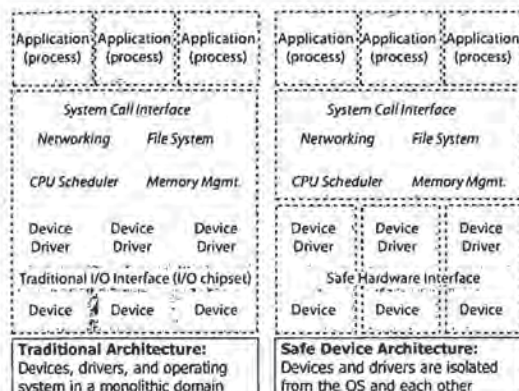


Figure 1: Isolation in a traditional OS (*lhs*) versus the Safe Device approach (*rhs*) which isolates device drivers from the kernel and from each other, and uses a *safe hardware interface* to extend this isolation to the device level.

drivers may leak memory, deadlock, fail to correctly manage interrupts, or wedge the system inside an infinite loop.

The broad spectrum of failure possibilities means that making systems dependable requires complete isolation of driver execution from that of the OS, applications and other drivers. More importantly, it is crucial that mechanisms be provided to ensure ongoing device availability, by recognising and reacting to driver failures. Our approach is to enforce *complete vertical isolation* of resources including device hardware, driver code, and operating system.

The difference between this new approach and traditional OS structure is shown in Figure 1 — device drivers are executed in a protection domain which restricts their access to host memory, I/O instructions, device registers and interrupt lines. Section 3 discusses our use of I/O Spaces, the mechanism by which we achieve these vertically isolated slices through the system. In Section 4 we present a set of techniques to address the difficult problem of *recognising* driver failure, and go on to identify problems that simply cannot be solved with current hardware.

1.2 Maintainability

Although hardly a simple undertaking, isolating driver code for safety only addresses a symptom of a considerably broader, architectural problem. Today, the development of driver code is tightly coupled with the individual target operating system. Device vendors must maintain separate source trees for each OS that they hope to support. Moreover, the individual driver OS interfaces are incredibly complex; Swift et al identify more than four hundred interface points between the Linux kernel and driver

code [2]. This, combined with the fact that OS interfaces differ not only syntactically but also semantically (e.g., providing very different threading models and communication primitives) means that each driver requires developers with OS-specific expertise.

This coupling benefits no one, except as an inertial force behind established operating systems. Even Windows is victim to the interface problem: Microsoft recently announced that the move to a 64-bit OS will require a new version of every device driver [3]. The lack of a common driver-OS interface is also an inhibitor to new OSs.

Our approach is to unify the driver-OS interface and use the isolation techniques discussed above to execute driver code in a completely separate execution context from that of the OS. The OS is presented with an idealized interface that describes a class of hardware, for instance storage devices or network interfaces. This approach allows a single driver to be used under any number of operating systems — each OS-specific driver serves an entire class of devices and typically comprises just a few hundred lines of code.

1.3 Manageability

Unified access to drivers is not just needed from a data access perspective, but also for administration. Presently, the tasks of diagnosing and configuring hardware are specific to each driver and OS instance. This has led, for instance, to specific device functionality being exposed under Windows but not Linux, and vice versa. The ad hoc means by which devices are currently administered equates to wasted administrator time and all the costs that entails. By unifying the control of devices in addition to their access, we hope to address these administrative concerns.

2 Related Work

The current I/O architecture presents a multifaceted set of challenging problems. This section attempts to summarize the great breadth of previous work that has attempted to tackle individual aspects of the problem. We have drawn on many of these efforts in our own research. There are two broad classes of work related to our own. First is a large set of efforts both in systems software and hardware development toward safe isolation. Second are attempts to better structure the interfaces between devices and their software, and the OSs and applications they interact with.

2.1 Safe Isolation

Researchers have long been concerned with the inclusion of extension code in operating systems. Extensible operating systems [4, 5] explored a broad range of approaches to support the incorporation of foreign, possibly untrusted

code in an existing OS. Swift et al [2] leverage the experiences of extensibility, particularly that of interposition, to improve the reliability of Linux device drivers. While their work claims an improvement in system reliability it demonstrates the risk of a narrow focus: their approach sacrifices performance drastically in an attempt to add dependability without modifying the existing OS. By addressing the larger architectural problem and not fixating on a single OS instance, we provide higher performance and solve a broader set of issues, while still remaining compatible with existing systems.

Our implementation, presented in Section 4, uses a virtualization layer to achieve isolation between drivers and the OS (or OSs) that use them. Providing a low-level systems layer that is principally responsible for managing devices was initially explored in Nemesis [6] and the Exokernel [7]. Our work refines these approaches by applying them to existing systems. Additionally, Whitaker et al [8] speculate as to the potential uses of a virtualized approach to system composition, drawing strongly on early microkernel efforts in Mach [9] among others [10, 11]. Our work represents a realization of these ideas, demonstrating that isolation can be provided with a surprisingly low performance overhead.

Commercial offerings for virtualization, such as VMWare ESX Server [12], allow separate OSs to share devices. While we have previously demonstrated [13] that our approach to virtualization provides higher performance, this work moves to focus specifically on additional concerns such as driver dependability; our implementation is now not only faster but also accommodates a strictly higher level of driver dependability.

Several research efforts have investigated hardware-assisted approaches to providing isolation on the PC platform. The Recovery-Oriented Computing [14] project, whose goals are similar to our own, have used hardware for system diagnostics [15], but defer to 'standard mechanisms' for isolation. Intel's SoftSDV [16], which is a development environment for operating systems supporting the IA-64 instruction set, uses PCI riser cards to proxy I/O requests. While their concern is in mapping device interrupts and DMA into the simulated 64-bit environment, the same approach could be used to provide device isolation. Intel has also announced that their new LaGrande architecture [17] will protect memory from device DMA.

2.2 Better Interfaces

Our goal of providing more rigid OS-device interfaces is hardly new. Most notably, corporate efforts such as UDI [18] have attempted to do just this. There are two key limitations of UDI that we directly address. Firstly, we enforce isolation whereas UDI-compliant drivers still execute in the same protection domain as the operating system, and

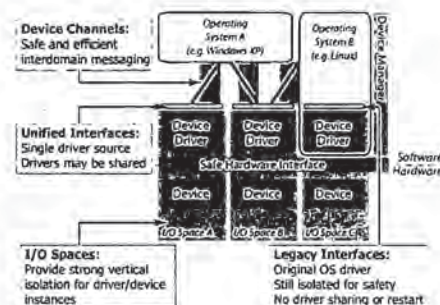


Figure 2: Example of the next-generation I/O architecture.

thus there is no mitigation of the risks posed by erroneous drivers. Secondly, our external perspective avoids the trap to which vendor consortiums such as UDI often fall victim: that of 'interface unioning'. Rather than providing the aggregate interface present in all existing drivers, we settle on a narrower, *idealized* interface. While we provide mechanisms to directly (and safely) expose the hardware should our interface be too constrictive, we have not found this to be a problem in our experiences with a large number of network and storage devices and several OSs.

Novel OS architectures have long struggled with a lack of device driver support. The vast number of available devices has compounded this problem, making the adoption of an existing driver interface attractive for fledgling systems. Microkernel systems such as Fluke [19] and L4 [20] have investigated wrapping Linux device drivers in customized interfaces [21, 22]. Although the structure of our architecture is not entirely dissimilar to that of a microkernel, our intent is to solve the driver interface issue for all operating systems on the PC architecture, rather than make some set of existing drivers work for a single developmental OS.

The LinuxBIOS project [23] replaces the proprietary BIOS on systems with a specialized Linux image. This approach allows fast startup and eases management, especially in cluster environments where console access is not available. This is closely related to our device control interface which we intend should be directly integrated with the BIOS.

3 Architecture

In this section we outline a new architecture that addresses the dependability, maintainability and manageability of devices and their control software. Note that we incorporate hardware modifications where they are necessary or desirable, deferring until Section 4 a more pragmatic design that allows for the limitations of existing systems.

Requirement 1: Driver Isolation	
Memory:	execute in logical fault domain
CPU:	schedule to prevent excessive consumption
Privilege:	limit access to instruction set
Requirement 2: Driver → Device Isolation	
I/O Registers:	restrict access to permitted ranges
Interrupts:	allow to mask/receive only device's interrupt
Requirement 3: Device Isolation	
Memory:	prevent DMA to arbitrary host memory
Other Devices:	prevent access to arbitrary other devices

Table 1: Requirements for Safe Hardware

As illustrated by Figure 2, our architecture comprises three parts which correspond directly to the problems identified. Firstly, we introduce *I/O Spaces* which arrange that devices perform their work in *isolation* from the rest of the system. This increases reliability by restricting the possible harm inflicted by device faults. Secondly, we define a set of per-class *unified interfaces* that are implemented by all devices of a particular type. This provides driver portability, avoiding the need to reimplement identical functionality for a range of different OS interfaces. Finally, our device manager provides a consistent *control and management* interface for all devices, simplifying system configuration and diagnosis and treatment of device problems.

3.1 Isolation

One reason for the catastrophic effect of driver failure on system stability is the total lack of isolation that pervades device interactions on commodity systems. The issues that must be addressed to achieve full isolation are outlined in Table 1. The concerns are divided into three requirements: isolating the execution of driver code from other software components, ensuring that drivers may only access the device they manage, and enforcing safe device behaviour.

Previous attempts at driver isolation [2] have placed driver code in a separate logical fault domain, essentially providing virtual memory protection between the driver and the rest of the system. However, this is only a partial solution as it primarily protects memory; a logical isolation layer *must* be used to provide isolation of scheduling and access to privileged instructions.

The implementation that we present in Section 4 uses a virtual machine monitor (VMM) to achieve the required logical isolation between driver and OS code. By tracking and retaining full control of each driver's CPU and memory use, the VMM provides isolation guarantees analogous to an OS and its application processes. For example, if a faulty driver becomes livelocked, or attempts to access a memory location outside its heap, then this is disallowed by the VMM and signalled to the device manager (described in Section 3.3) which takes appropriate remedial action.

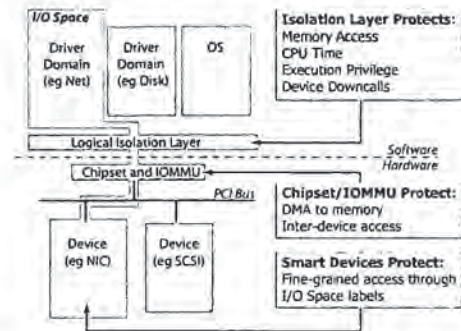


Figure 3: Achieving Safe Hardware. The outlined region denotes an *I/O Space*: a vertical slice through the system providing isolation for device and driver.

Importantly, our approach also addresses the problem of isolating physical device access. Currently, drivers may write to arbitrary device registers or mask inappropriate interrupts, and devices may DMA to invalid memory. We introduce *I/O Spaces*, which extend the notion of logical protection domains to incorporate resources specific to devices and their driver code (Figure 3).

An *I/O Space* is a vertical slice through the system, providing an isolated context for each device and its driver. A controller within the *I/O* chipset maintains tables of access permissions for each *I/O Space*, identifying accessible ranges of memory addresses, device registers and interrupt lines. Each *I/O Space* is represented by a numeric identifier that is attached to every *I/O* transaction by a device or driver operating within that Space. The controller uses this identifier to validate the requested operation against the appropriate permission table.

Further advantages can be gained by allocating a range of *I/O Spaces* to each device. Incorporating a notion of *client identity* into each *I/O Space* would enable features such as client scheduling within the device; and safe DMA to application buffers [24]. It is also a small step to predicate bus arbitration on the requesting *I/O Spaces*, thus allowing differentiated service to be provided to different clients.

As the implementation that we present in Section 4 uses virtualization, we have been able to address the physical isolation problems of host-to-device access by implementing within the VMM the *I/O-Space* functionality of a next-generation chipset. We believe that the isolation we have achieved is the strongest possible without hardware modifications. Although our current implementation cannot protect against unsafe device DMA, we describe the minor modifications that would be necessary to take advantage of a safe DMA controller. Emerging hardware research [15, 16, 17] indicates that these hardware improvements may soon be incorporated into the PC platform.

3.2 Unified Interfaces

Although the PC has standardized hardware interfaces there is no such accepted standard for the interface to system software, despite industry efforts [18]. Our solution is to define a set of idealized high-level interfaces tailored for each class of device. OS vendors then need implement only a single, small driver per *device class* that communicates via the unified interface: this can be developed in-house by developers with intimate knowledge of the OS, and subjected to appropriate quality-control checks. By implementing the unified interface, hardware vendors automatically support every PC system. Furthermore, they may arbitrarily choose how the implementation is divided between hardware and software, perhaps incorporating more functionality into higher-cost products that include advanced features such as I/O processors [25].

Our unified device interfaces are based on those provided by the Xen VMM which, as we have previously demonstrated [13], provides low-overhead access to common device classes. The essential features required for efficient data-path communication are to avoid data copies, to provide back pressure to the data source, and to use a flexible and asynchronous notification primitive. Within our architecture we incorporate these principles into *device channels*, linking the unified interfaces exported by device drivers to the operating systems using them. We provide details of a software implementation of device channels in Section 4.2. However, we were careful in our design not to exclude the possibility of a hardware implementation.

Concerns regarding the feasibility of adopting standardized device interfaces are very relevant, as acceptance is more of a political problem than a technical one. Our efforts to date have had a great deal of success in allowing a variety of networking and storage devices to function through a common interface to Linux, NetBSD, and Windows XP. We have focused on these classes of device as we believe that network and disk are the two most crucial device interfaces in a server environment. We do not presume that the interfaces we have identified are complete, and expect them to evolve over time. However, experience so far has shown that our model is valid; other groups (e.g. [26]) have independently ported new devices to our architecture with minimal effort.

The end-to-end argument [27] has been invoked by efforts in the past, particularly Exokernels [7], as motivation to expose rather than to abstract hardware interfaces. While we believe that unified interfaces provide considerable benefit, we must also acknowledge that it is likely impossible to effectively model all devices: emerging devices and special-purpose applications must be considered. In these situations, we allow device access to be exposed directly, and it is through this mechanism that we address video and sound devices in our current implementation. Note that

even when we do not use a unified virtualized device interface, the architecture still provides isolation and safety. This transitional approach allows our architectural benefits to be realized in the short term, while we move to focus on the challenging problems of sound and video interfaces in the future.

It is additionally worth observing that organisations continue to move toward OS virtualization as a means of making better use of server hardware. Unified interfaces are particularly advantageous in a virtualized environment where they can enable *device sharing*.

An example of unified interfaces, legacy support, and device sharing was shown in Figure 2 in which two operating systems and three device drivers all run on a single machine. The two leftmost device drivers present a unified interface which ‘wraps’ existing driver code. Using this interface means that device drivers may be individually scheduled, shared between operating systems, and restarted in case of error. The rightmost operating system contains a legacy driver; although this prevents separate scheduling or sharing, the safe hardware interface can still be used to limit the driver’s privileges.

3.3 Control and Management

The final concern addressed by our architecture is that of device control and configuration — an area that has been particularly neglected during the PC’s evolution. The lack of standardized platform-wide control interfaces has led to the implementation of unique and proprietary configuration interfaces for each OS and device¹. A significant disadvantage of this ad hoc approach is that system administrators require additional training for each OS environment and machine setup that they support, simply to understand multiple different configuration interfaces that ultimately provide identical functionality.

The transition of the PC platform into the server room means that manageability is now more important than ever. The current jumble of configuration tools is inappropriate for configuring and managing the large-scale clusters that are common in enterprise environments. Console-based interfaces, although suitable for configuring small numbers of desktop machines, are a major hindrance when configuration changes must be applied to hundreds of machines at a time. The growing problem of remote management is a primary motivation for the LinuxBIOS project [23].

This final aspect of our architecture is handled by a *device manager* — essentially an extension to the system BIOS that provides a common set of management interfaces for all devices. The device manager is responsible for boot-

¹ Some common device classes do enjoy a consistent control interface, but even this consistency is not carried across different OSs.

strapping isolated device drivers, announcing device availability to OSs, and exporting configuration and control interfaces to either a local OS or to a remote manager.

4 Design and Implementation

We have implemented our next-generation I/O architecture for current PC hardware, based on the Xen virtual machine monitor. As described in [13], Xen divides the resources of a PC system amongst a set of secure and performance-isolated *domains*, each of which runs a separate guest operating system and applications. Xen implements only isolation *mechanisms*: management tasks, such as domain creation and resource allocation, are performed by a *system controller* running in a special domain with access to a privileged control interface.

We begin this section by describing how we extended Xen's virtual-machine and control interfaces to allow safe access to hardware. By placing device drivers in a resource-controlled domain separate from OS and application code, configured with suitably restrictive hardware-access privileges, Xen provides the isolation of processor and hardware contexts that we identified in requirements one and two of Section 3.1. We incorporate the necessary control and management services, provided by a *device manager* in our architectural outline, into a device-management subsystem of the system controller.

We then proceed to describe how guest OSs *connect* to drivers in other domains. We introduce an efficient method for inter-domain communication based on shared memory and asynchronous notifications, and outline the protocol for setting up *device channels* using a core interface that links every domain to the system controller. Device channels provide a unified abstraction for high-performance data transfer: we describe how this abstraction is safely implemented by device drivers and used by guest OSs.

4.1 Safe Hardware Interface

Our safe hardware interface enforces isolation of device drivers by restricting the hardware resources that they can access. To this end, we restrict access privileges to device I/O registers (whether memory-mapped or accessed via explicit I/O ports) and interrupt lines. Furthermore, where it is possible within the constraints of existing hardware, we protect against device misbehaviour by isolating device-to-host interactions. Finally, we virtualize the PC's hardware *configuration space*, allowing the system controller unfettered access so that it can determine each device's resources, while restricting each driver's view of the system so that it cannot see resources that it cannot access.

4.1.1 I/O Registers

Xen ensures memory isolation amongst domains by checking the validity of address-space updates. Access to a memory-mapped hardware device is permitted by extending these checks to allow access to non-RAM page frames that contain memory-mapped registers belonging to the device. Page-level protection is sufficient to provide isolation because register blocks belonging to different devices are usually aligned on no less than a page boundary.

In addition to memory-mapped I/O, many processor families provide an explicit I/O-access primitive. For example, the x86 architecture provides a 16-bit I/O port space to which access may be restricted on a per-port basis, as specified by an access bitmap that is interpreted by the processor on each port-access attempt. Xen uses this hardware protection by rewriting the port-access bitmap when context-switching between domains. Since the bitmap is large and sparse, for each domain Xen tracks and rewrites only the active words within the bitmap.

4.1.2 Interrupts

Whenever a device's interrupt line is asserted it triggers execution of a stub routine within Xen rather than causing immediate entry into the domain that is managing that device. In this way Xen retains tight control of the system by *scheduling* execution of the domain's interrupt service routine (ISR). Taking the interrupt in Xen also allows a timely acknowledgement response to the interrupt controller (which is always managed by Xen) and allows the necessary address-space switch if a different domain is currently executing. When the correct domain is scheduled it is delivered an asynchronous *event notification* which causes execution of the appropriate ISR.

Xen notifies each domain of asynchronous events, including hardware interrupts, via a general-purpose mechanism called *event channels*. Each domain can be allocated up to 1024 event channels, each of which comprises a pair of bit flags in a memory page shared between the domain and Xen. The first flag is used by Xen to signal that an event is *pending*. When an event becomes pending Xen schedules an asynchronous upcall into the domain; if the domain is blocked then it is moved to the run queue. Unnecessary upcalls are avoided by triggering a notification only when an event first becomes pending: further settings of the flag are then ignored until after it is cleared by the domain.

The second event-channel flag is used by the domain to *mask* the event. No notification is triggered when a masked event becomes pending: no asynchronous upcall occurs and a blocked domain is not woken. By setting the mask before clearing the pending flag, the domain is able to prevent unnecessary upcalls for partially-handled event sources. When the mask is eventually cleared the domain

can reread the pending flag to see whether another batch of work has arrived from the event source.

Each domain specifies a single upcall handler for all event-channel notifications. To avoid the expense of linearly scanning all pending flags, a *selector word* indicates which aligned groups of 32 channels are pending. This two-level hierarchy permits fast scanning in the common situation that few channels are pending.

To avoid unbounded reentrancy, a level-triggered interrupt line must be masked at the interrupt controller until all relevant devices have been serviced. Because of this, after handling an event relating to a level-triggered interrupt, the domain must call *down* into Xen to unmask the interrupt line. However, if an interrupt line is not shared by multiple devices then Xen can usually safely reconfigure it as edge-triggering. This obviates the need for unmask downcalls, as they are not required for edge-triggered interrupt lines.

When an interrupt line is shared by multiple hardware devices, Xen must delay unmasking the interrupt until a downcall is received from every domain that is managing one of the devices. Xen cannot guarantee perfect isolation of a domain that is allocated a shared interrupt: if the domain never unmask the interrupt then other domains can be prevented from receiving device notifications. However, shared interrupts are rare in server-class systems which typically contain IRQ-steering and interrupt-controller components with enough pins for every device. The problem of sharing is set to disappear completely with the introduction of message-based interrupts as part of PCI Express [28].

4.1.3 Device-to-Host Interactions

As well as preventing a device driver from circumventing its isolated environment, we must also protect against possible misbehaviour of the hardware itself, whether due to inherent design flaws or misconfiguration by the driver software. The two general types of device-to-host interaction that we must consider are assertion of interrupt lines, and accesses to host memory space.

Protecting against arbitrary interrupt assertion is not a significant issue because, except for shared interrupt lines, each hardware device has its own separately-wired connection to the interrupt controller. Thus it is physically impossible for a device to assert any interrupt line other than the one that is assigned to it. Furthermore, Xen retains full control over configuration of the interrupt controller and so can guard against problems such as 'IRQ storms' that could be caused by repeated cycling of a device's interrupt line.

The main 'protection gap' for devices, then, is that they may attempt to access arbitrary ranges of host memory. For example, although a device driver is prevented from using the CPU to write to a particular page of system memory

(perhaps because the page does not belong to the driver), it may instead program its hardware device to perform a DMA to the page. Unfortunately there is no good method for protecting against this problem with current hardware as it is infeasible for Xen to validate the programming of DMA-related device registers. Not only would this require intimate knowledge of every device's DMA engine, it also would not protect against bugs in the hardware itself: buggy hardware would still be able to access arbitrary system memory.

A full implementation of this aspect of our design requires integration of an IOMMU into the PC chipset. Similar to the processor's MMU, this translates the addresses requested by a device into valid host addresses. Inappropriate host addresses are not accessible to the device because no mapping is configured in the IOMMU. In our design, Xen would be responsible for configuring the IOMMU in response to requests from domains. The required validation checks are identical to those required for the processor's MMU; for example, to ensure that the requesting domain owns the page frame, and that it is safe to permit arbitrary modification of its contents.

It is entirely reasonable to expect that IOMMU functionality will be included in commodity chipsets in the near future: 64-bit systems already include an IOMMU that allows 32-bit devices to access the full range of host memory. Although the IOMMU is intended to perform only translation, and devices are not prevented from bypassing its memory window, it is a small step to require that all memory transactions pass through the IOMMU and thus also use it to enforce protection.

4.1.4 Hardware Configuration

The PCI standard defines a generic *configuration space* through which PC hardware devices are detected and configured. Xen restricts each domain's access to this space so that it can only read and write registers belonging to a device that it owns. This serves a dual purpose: not only does it prevent cross-configuration of other domains' devices, but it also restricts the domain's view so that a hardware probe detects only devices that it is permitted to access.

The method of access to the configuration space is system-dependent, and the most common methods are potentially unsafe (either protected-mode BIOS calls, or a small I/O-port 'window' that is shared amongst all device spaces). Domains are therefore not permitted direct access to the configuration space, but are forced to use a virtualized interface provided by Xen. This has the advantage that Xen can perform arbitrary validation and translation of access requests. For example, Xen disallows any attempt to change the base address of an I/O-register block, as the new location may conflict with other devices.

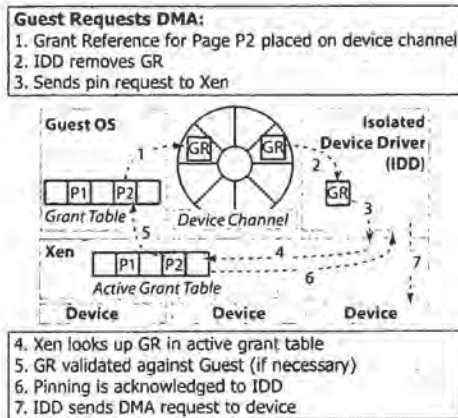


Figure 4: Using device channel to request a data transfer.

The system controller is permitted access to the entire configuration space. This eases configuration of the safe hardware interface because the configuration space comprehensively describes the hardware resources that belong to each device. Thus the controller can automatically configure the correct access permissions for a new device-driver domain without assistance from the system administrator.

4.2 Device Channels

Although the safe hardware interface can be configured to allow a guest OS to run its own device drivers, this misses the potential improvements in reliability, maintainability and manageability of running device drivers in isolation. It is therefore expected that each device driver will run within an isolated driver domain (IDD) which limits the impact of driver faults.

Guest OSs access a device through a *device channel* link with its IDD. The channel is a point-to-point communication link through which each end can send messages asynchronously to the other. A device channel is established by using the system controller to introduce the IDD to the guest OS, and vice versa. To make this possible, the system controller automatically establishes an initial control channel with each domain that it creates. Figure 4 shows a guest OS requesting a data transfer through a device channel. The individual steps involved in the request are discussed later in this section.

Instead of treating IPC as a fundamental primitive, as in most compartmentalized systems [8, 20], Xen itself has no concrete notion of a control or device channel. Messages are communicated via shared memory pages that are allocated by the guest OS but are simultaneously mapped into the address space of the IDD or system controller. For this purpose, Xen permits restricted *sharing* of memory pages

between domains.

4.2.1 Sharing Memory

The sharing mechanism provided by Xen differs from traditional application-level shared memory in two key respects: shared mappings are *asymmetric* and *transitory*. Each page of memory is owned by at most one domain at any time and, with the assistance of Xen and the system controller, that owner may force reclamation of mappings from within other misbehaving domains.

To add a foreign mapping to its address space, a domain must present a valid *grant reference* to Xen in lieu of the page number. A grant reference comprises the identity of the domain that is granting mapping permission, and an index into that domain's *grant table*. Every domain owns a private grant table that it shares only with Xen, in which each entry is a tuple $(grant, D, P, R, U)$ permitting domain D to map page P into its address space; asserting the boolean flag R restricts D to read-only mappings. The flag U is written by Xen to indicate whether D currently maps P (i.e., whether the grant tuple is *in use*).

When Xen is presented with a grant reference (A, G) by a domain B , it first searches for index G in domain A 's *active grant table* (AGT), a private table that is only accessible by Xen. If no match is found, Xen reads the appropriate tuple from the guest's grant table and checks that $T=grant$ and $D=B$, and that $R=false$ if B is requesting a writeable mapping. Only if the validation checks are successful will Xen copy the tuple into the AGT and mark the grant tuple as in use.

Xen tracks uses of grant references by associating a usage count with each AGT entry. Whenever a foreign mapping is created with reference to an existing AGT entry, Xen increments that entry's count. The grant reference cannot be reallocated or reused by the granting domain until the foreign domain destroys all mappings that were created with reference to it.

Although it is clear that this mechanism allows strict checking of foreign mappings when they are created, it is less obvious how these mappings might be revoked. For example, if a faulty IDD stops responding to service requests then guest OSs could end up owning unusable memory pages. We handle the possibility of driver failure by taking a deadline-based approach: if a guest observes that a grant table entry is still marked as in use when it determines that it ought to have been relinquished (e.g., because it requested that the device channel should be destroyed), then it signals a potential domain failure to the system controller.

The system controller checks whether the specified grant reference exists in the notifying domain's AGT and, if so,

sets a deadline by which the suspect domain must relinquish the stale mappings. If a registered deadline passes but stale mappings still exist then Xen notifies the system controller. At this point the system controller may choose to destroy and restart the driver, thereby forcibly reclaiming the foreign mappings.

4.2.2 Descriptor Rings

I/O descriptor rings are used for asynchronous transfers between a guest OS and an IDD. Ring updates are based around two pairs of producer-consumer indexes: the guest OS places service requests onto the ring, advancing a request-producer index, while the IDD removes these requests for handling, advancing an associated request-consumer index. Responses are queued onto the same ring as requests, albeit with the IDD as producer and the guest OS as consumer. There is no requirement that requests be processed in order: the guest OS associates a unique identifier with each request which is reproduced in the associated response. This allows the IDD to unambiguously reorder I/O operations due to scheduling or priority considerations.

The guest OS and IDD use a shared *inter-domain* event channel to send asynchronous notifications of queued descriptors. An inter-domain event channel is similar to the interrupt-attached channels described in Section 4.1.2. The main differences are that notifications are triggered by the domain attached to the opposite end of the channel (rather than Xen), and that the channel is *bidirectional*: each end may independently notify the other, or mask incoming notifications.

We decouple the production of requests or responses on a descriptor ring from the notification of the other party. For example, in the case of requests, a guest may enqueue multiple entries before notifying the IDD; in the case of responses, a guest can defer delivery of a notification event by specifying a threshold number of responses. This allows each domain to independently balance its latency and throughput requirements.

4.2.3 Data Transfer

Although storing I/O data directly within ring descriptors is a suitable approach for low-bandwidth devices, it does not scale to high-performance devices with DMA capabilities. When communicating with this class of device, which includes fast network interfaces and disc arrays, data buffers are instead allocated out-of-band by the guest OS and indirectly referenced within I/O descriptors.

When programming a DMA transfer directly to or from its hardware device, the IDD must first *pin* the data buffer. As described in Section 4.2.1, we enforce driver isolation by requiring the guest OS to pass a grant reference in lieu of

the buffer address when requesting a device transfer: the IDD specifies this grant reference when pinning the buffer. Xen applies the same validation rules to pin requests as it does for address-space mappings. These include ensuring that the memory page belongs to the correct domain, and that it isn't attempting to circumvent memory-management checks (for example, by requesting a device transfer directly into its page tables).

Returning to the example in Figure 4, the guest's data-transfer request includes a grant reference *GR* for a buffer page P_2 . The request is dequeued by the IDD which sends a pin request, incorporating *GR*, to Xen. Xen reads the appropriate tuple from the guest's grant table, checks that P_2 belongs to the guest, and copies the tuple into the AGT. The IDD receives the address of P_2 in the pin response, and then programs the device's DMA engine.

On systems with protection support in the chipset (Section 4.1.3), pinning would trigger allocation of an entry in the IOMMU. This is the only modification required to enforce safe DMA (Requirement 3 in Table 1). Moreover, this modification affects only Xen: the IDDs are unaware of the presence or otherwise of an IOMMU (in either case pin requests return a bus address through which the device can directly access the guest buffer).

In addition to pinning a guest buffer for DMA, an IDD may also use the grant reference to map the buffer into its address space. This allows support of legacy devices which do not support DMA; the IDD instead uses the main CPU to transfer data to or from the device. Buffer mapping is also useful for network-driver domains, which may choose to copy each packet header into a guest-inaccessible buffer before applying filtering rules. In this case the driver will usually both map and pin the packet buffer: all modern network interfaces support scatter-gather DMA, so they can transfer the packet payload directly from the guest buffer.

4.2.4 Device Sharing

Since Xen can simultaneously host many guest OSs it is essential to consider issues arising from device sharing. The control mechanisms for creating and destroying device channels naturally support multiple channels to the same IDD. In this section we describe how our block-device and network IDDs support multiplexing of service requests from different clients.

Within our block-device driver we service *batches* of requests from competing guests in a simple round-robin fashion; these are then passed to a standard elevator scheduler before reaching the disc controller. The low-level scheduling provided by the elevator, and also by many disc controllers, gives us good throughput, while request batching provides reasonably fair access. We take a similar approach

for network transmission, where we implement a credit-based scheduler allowing each device channel to be allocated a bandwidth share of the form x bytes every y micro-seconds. When choosing a packet to queue for transmission, we round-robin schedule amongst all the channels that have sufficient credit.

Sharing a high-performance network-receive path requires careful design because, apart from a few smart network interfaces that perform packet demultiplexing in hardware [24], it is not possible to DMA directly into a guest-supplied buffer. Rather than copying the packet into a guest buffer after performing demultiplexing, we instead *exchange ownership* of the page containing the packet with an unused page provided by the guest OS. This avoids copying overheads but requires the IDD to queue page-sized buffers at the network interface. When a packet is received, the IDD immediately checks its demultiplexing rules to determine the destination device channel. If no unused pages are queued on the guest's network-receive ring then the packet is dropped.

For safety, Xen does not permit IDDs to exchange ownership of arbitrary memory pages. Instead we extend the grant table (Section 4.2.1) to include *exchange tuples*, (*exchange*, D , P , Q), permitting domain D to acquire ownership of page P in exchange for relinquishing ownership of another page. Unused exchange tuples are denoted by $Q=0$; when the table entry is used, Xen rewrites Q with the address of the page that was relinquished by the IDD. When exchanging page ownerships, the IDD sends a request *exchange*(P , G) to Xen, where P is a page frame belonging to the IDD, and G is an *exchange grant* passed to the IDD by the guest OS in a network-receive descriptor.

4.3 Control and Management

We embed our device manager within the system controller: a small privileged management kernel that is loaded from firmware when the system boots. During bootstrap, the device manager probes device hardware and creates an IDD, loaded with the appropriate driver, for each detected device. The device manager's ongoing responsibilities then include per-guest device configuration, managing setup of device channels, providing interfaces for hardware configuration, and reacting to driver failure.

4.3.1 Guest Configuration and Bootstrap

The device manager extends the domain-management functions of the system controller by allowing configuration of restricted IDD access for each guest OS. For example, a network-device channel may be prevented from sending packets with a spoofed source address, or a block-device channel may be limited to isolated regions of a shared disc.

As each guest OS boots, the manager informs it of the devices to which it has been granted access. The guest OS then initiates device-channel creation by allocating a memory page for the I/O-communications ring and passing a grant reference to the IDD via the device manager. The manager allocates an inter-domain event channel linking the guest OS to the IDD, and passes one endpoint of this channel, together with the grant reference, to the IDD. When the IDD acknowledges setup of the device channel, the response is forwarded to the guest OS together with its event-channel endpoint.

4.3.2 Driver Failure and Restartability

In our design, the device manager is responsible for detecting driver failure and coordinating recovery. There are several ways in which the manager may determine that a driver has failed: for example, it may receive notification from Xen that the IDD has crashed, or an unresponsive IDD may fail to unmap or unpin guest buffers within a specified time period. The subsequent recovery phase is greatly simplified by the componentized design of our I/O architecture: firstly, the shared state associated with a device channel is small and well-defined; and secondly, IDD-internal state is 'soft' and therefore may simply be reinitialized when it restarts.

The recovery phase comprises several stages. First, the device manager destroys the offending IDD and replaces it with a freshly-initialized instance. The manager then signals to the connected guest OSs that the IDD has restarted; each guest is then responsible for connecting itself to the new device channel, using the normal signalling mechanisms provided by the device manager. At this point, the guest may also opt to reissue requests that may have been affected by the failure (i.e., outstanding requests for which no response was received before the IDD failed).

We have implemented OS-specific network and block-device drivers that are able to recover from an IDD restart. During recovery, the drivers retain a small amount of state to reestablish channel connections and reissue incomplete requests. State held within the shared device-channel memory (which may have been corrupted by the failed IDD) is discarded, increasing our tolerance to potential driver faults.

5 Evaluation

In this section we present an evaluation of our prototype implementation. We begin by evaluating the impact of our isolation mechanisms on realistic application workloads using industry standard benchmarks such as Postmark [29].

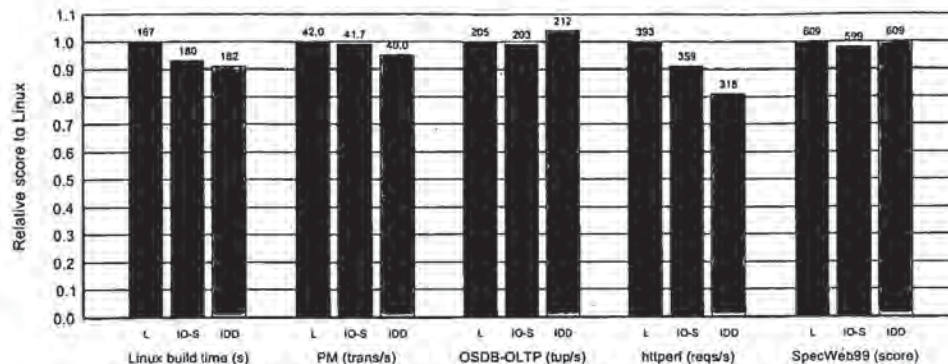


Figure 5: Application-Level Benchmarks. (L=L-SMP, IO-S=IO-Space, IDD=IDD-SMP)

SPEC WEB99² and OSDB³. We then proceed to investigate our overheads on individual device subsystems using a series of network and disk micro-benchmarks to determine the overhead of implementing the I/O Space protected hardware interface, and the additional overhead of device-driver isolation via the device channel interface. Finally, we provoke a series of device-driver failures and measure system availability while recovering.

All experiments were performed on a Dell PowerEdge 2650 dual processor 3.06Ghz Intel Xeon server with 1GB of RAM, two Broadcom Tigon 3 Gigabit Ethernet network cards, and an Adaptec AIC-7899 Ultra160 SCSI controller with two Fujitsu MAP3735NC 73GB 10K RPM SCSI disks. Linux version 2.4.26 and RedHat 9.0 Linux were used throughout, installed on an ext3 file-system. Identical device driver source code from Linux 2.4.26 is used throughout our experiments, allowing us to measure only those performance variations that are caused by varying the I/O system configuration.

Our current implementation of Isolated Driver Domains (IDDs) is based on the Xen virtual machine monitor. Xen's paravirtualized interface results in a very low virtualization overhead for CPU and memory intensive applications. When running benchmarks like SPEC CPU2000 over Xen, the measured overhead is less than 1% [13]. In the following experiments we can therefore attribute any slow down relative to native Linux to our I/O modifications. We believe this establishes a good upper bound on the additional costs, but note that the overhead may reduce if future x86 processors provide hardware assistance to reduce the cost of virtualization, or if client-aware devices reduce the load on the host processor.

We compare the performance of our IDD prototype against a number of other configurations, using a vanilla Linux 2.4.26 SMP kernel as our baseline (L-SMP). To estab-

lish the overhead of implementing protected hardware access we measure a version of Xen/Linux containing disk and network drivers that access the hardware via the protected interface, which provides virtualization of interrupts and segregation of hardware access. We label these results IO-Space. We also evaluate the performance of our full-blown architecture using IDD for each of the network and disk devices, communicating with an instance of Xen/Linux using device-channel I/O interfaces. Each IDD and Xen/Linux instance runs in its own isolated Xen domain; the CPU to which each domain is bound depends on the test configuration. We include results in which the Xen/Linux instance executes on the same CPU as the IDDs (IDD-UP), in which the Xen/Linux instance runs on a different physical CPU (IDD-SMP), and in which the Xen/Linux instance is bound to a different 'hyper thread' within the same physical CPU (IDD-HT).

5.1 Application-Level Benchmarks

We subjected our test systems to a battery of application-level benchmarks, the results of which are displayed in Figure 5. Our first benchmark measures the elapsed time to do a complete build of the default configuration of a Linux kernel tree stored on the local ext3 file system. The kernel compile performs a moderate amount of disk I/O as well as spending time in the OS kernel for process and memory management, which typically introduces some additional overhead when performed inside a virtual machine. The results show that the I/O Space virtualized hardware interface incurs a penalty of around 7%, whereas the full IDD architecture exhibits a 9% overhead.

Postmark is a file system benchmark developed by Network Appliance which emulates the workload of a mail server. It initially creates a set of files with varying sizes (2000 files with sizes ranging from 500B to 1MB) and then performs a number of transactions (10000 in our configuration). Each transaction comprises a variety of operations including file

²<http://www.spec.org/osg/web99/>

³<http://osdb.sourceforge.net/>

creation, deletion, and appending-write. During each run over 7GB of data is transferred to and from the disk. Postmark reports three figures for each configuration: the number of transactions per second, and the aggregate read and write throughputs. Since the relative results for all three metrics are very similar, we present only transaction rates. The additional overhead incurred by I/O Spaces and the full IDD architecture are just 1% and 5% respectively.

OSDB is an Open Source database benchmark that we use in conjunction with PostgreSQL 7.3.2. The benchmark creates and populates a database and then, in multi-user mode, exercises two types of workload: Information Retrieval (IR), which performs read-only operations over the entire database; and On-Line Transaction Processing (OLTP), which both queries and updates tuples in the database. As the default dataset of 40MB fits entirely into the buffer cache, we created a dataset containing one million tuples per relation, resulting in a 400MB database. We only present the results for the OLTP workload as IR does not generate significant disk activity. We investigate the surprisingly high result achieved by IDD in our disk microbenchmark in Section 5.3.

httpperf-0.8 was used to generate requests to an Apache 2.0.40 server to retrieve a single 64kB static HTML document. The benchmark was configured to maintain a single outstanding HTTP request, thus effectively measuring the response time of the server. The resulting network bandwidth generated by the server is around 200Mb/s. The I/O Space result exposes the overhead of virtualizing interrupts in this latency-sensitive scenario in which there is no opportunity to amortise the overhead by pipelining requests. Communicating with the IDD via the device channel interface compounds the effect by requiring a significant number of inter-domain notifications. Despite this, the response time is within 19% of that achieved by native L-SMP.

SPEC WEB99 is a complex application-level benchmark for evaluating web servers and the systems that host them. The workload is a complex mix of page requests: 30% require dynamic content generation, 16% are HTTP POST operations and 0.5% execute a CGI script. As the server runs it generates access and POST logs, so the disk workload is not solely read-only. During the measurement period there is up to 200Mb/s of TCP network traffic and considerable disk read-write activity on a 2.7GB dataset.

A number of client machines are used to generate load for the server under test, with each machine simulating a collection of users concurrently accessing the web site. The benchmark is run repeatedly with different numbers of simulated users to determine the maximum number that can be supported. *SPEC WEB99* defines a minimum Quality of Service that simulated users must receive in order to be 'conformant' and hence count toward the score: after an initial warm-up phase, users must receive an aggregate

bandwidth in excess of 320Kb/s over a series of requests.

For our experimental setup we used the Apache HTTP server version 1.3.27 with the *modspecweb99* plug-in to perform most of the dynamic content generation (SPEC rules require 0.5% of requests to use full CGI, forking a separate process). Under this demanding workload we find that the overhead of I/O Spaces and even full device driver isolation to be minimal: just 1% and 2% respectively.

5.2 Network performance

We evaluated the network performance of our test configurations by using *netp* to measure TCP throughput over Gigabit Ethernet to a second host running L-SMP. Both hosts were configured with a socket buffer size of 128KB as this is recommended practice for Gigabit networks. We repeated the experiment using two different MTU sizes, the default Ethernet MTU of 1500 bytes, and a smaller MTU of 552 bytes. The latter was picked as it is commonly used by dial-up PPP clients, and puts significantly higher stress on the I/O system due to the higher packet rates generated (190,000 packets a second at 800Mb/s).

Using a 1500 byte MTU all configurations achieve within a few percent of the maximum throughput of the Gigabit Ethernet card, which is the system bottleneck (Table 2). The 552 byte MTU provides a far more demanding test, exposing the different per-packet CPU overheads between the configurations. The virtualized interrupt dispatch model provided by I/O Spaces incurs an overhead of 11% on transmit and 5% on receive. This shows that, even under extreme load, retaining safe control of interrupt dispatch and device access can be achieved at reasonable cost.

The figures for our IDD implementation reflect the extra CPU cost of full driver isolation. The single CPU result represents close to a worst case scenario, recording performance slow downs of around 50% relative to L-SMP. This reflects the cost of rapid switching between protection domains and the deleterious effect that this has on the cache and TLB. Enabling hyper-threading (IDD-HT) on our single-CPU configuration provides some relief, avoiding the context switching and allowing a better data-flow through the processor's cache. Adding a second CPU (IDD-SMP) provides further benefits, reducing the relative deficit to 3% on transmit and 18% on receive.

5.3 Disk performance

Unlike networking, disk I/O typically does not impose a significant strain on the CPU because data is typically transferred in larger units and with less per-operation overhead. We performed experiments using *dd* to repeatedly write and then read a 4 GB file to and from the same ext3 file system (Table 3). Read performance is nearly identical

	TCP MTU 1500		TCP MTU 552	
	TX	RX	TX	RX
L-SMP	897	897	808	808
I/O Space	897 (0%)	898 (0%)	718 (-11%)	769 (-5%)
IDD-UP	897 (0%)	843 (-5%)	436 (-46%)	379 (-53%)
IDD-HT	897 (0%)	897 (0%)	651 (-19%)	577 (-29%)
IDD-SMP	897 (0%)	898 (0%)	778 (-3%)	663 (-18%)

Table 2: *tcp*: Bandwidth in Mb/s

	read	write
L-SMP	66.01	47.36
I/O Space	65.78 (-0%)	46.74 (-1%)
IDD-SMP	65.16 (-1%)	58.47 (+23%)

Table 3: *dd*: Bandwidth in MB/s

in all cases, but attempts to measure write performance are hampered due to an oscillatory behaviour of the Linux 2.4 memory system when doing bulk writes. This leads to our IDD configurations actually outperforming standard Linux as the extra stage of queueing provided by the device channel interface leads to more stable throughput.

5.4 Device Driver Recovery

In these tests we provoked our network driver to perform an illegal memory access, and then measured the effect on system performance. In this scenario detection of the device driver failure is immediate, unlike internal deadlock or infinite looping where there will be a detection delay dependent on system timeouts.

To test driver recovery we caused an external machine to send equally-spaced ping requests to our test system at a rate of 200 packets per second. Figure 6 shows the inter-arrival latencies of these packets at a guest OS as we inject a failure into the network driver domain at 10-second intervals. During the recovery period after each failure we recorded network outages of around 275ms. Closer examination revealed that much of this period is spent executing the device driver's media detection routines while determining the link status.

During driver restart, packets that are received at the network interface are lost. A 275ms average recovery time can interact with TCP's congestion control mechanism to cause a longer effective interruption to service. We have performed experiments running full-rate TCP connections between two hosts while restarting the network driver. In this scenario we observed the TCP connection begin RTO-triggered retransmissions during the outage, effectively adding additional time to complete system recovery.

We have repeated these experiments using a 'hot standby' driver domain that resets the network interface and re-advices device channels when signalled by the system con-

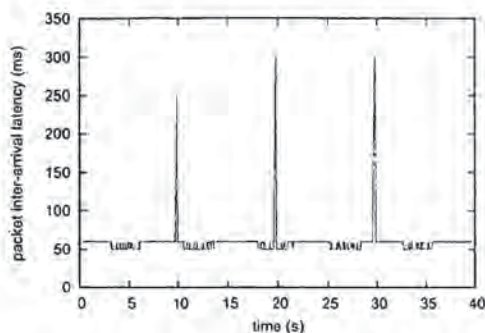


Figure 6: Effect of driver restart on packet arrivals.

troller. This approximately halves the network outage. Since each driver domain requires just 3MB of memory, this solution may be attractive in some scenarios.

We have also conducted similar experiments restarting block device drivers. The driver downtime in such experiments is largely determined by the time to scan the SCSI bus for devices. In situations where this is known in advance the driver could be restarted with a specific list of devices, avoiding the scan time.

6 Conclusion

We have presented a next-generation I/O architecture which solves existing problems of dependability, maintainability and manageability. Key to achieving this is the separation of device drivers from operating systems; by running each device driver in a separately protected and scheduled environment, we increase the robustness of systems to bugs in hardware and software. By using unified device driver interfaces we can *share* devices between a number of co-existing operating system instances, and dynamically *restart* device drivers in case of error or upgrade.

Unified interfaces also increase portability, allowing different kinds of operating systems to use the same device-specific code; the operating system need provide only a simple generic driver for an entire class of devices. Even when generic interfaces are inappropriate (e.g. for non-mainstream or non-shareable devices), the isolation we offer can still increase reliability and aid management.

Although the hardware required to fully support our I/O architecture is not yet available, we have implemented a prototype which makes use of a virtual machine monitor to provide the requisite functionality. The prototype supports nearly all the features of our architecture (a notable exception being protection against erroneous DMA), and gives surprisingly good performance – overhead is generally less than a few percent, and restartability can be achieved within

a few hundred milliseconds. Furthermore, we believe that our implementation can naturally incorporate and benefit from emerging hardware support for protection.

References

- [1] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, October 2001.
- [2] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 207–222, October 2003.
- [3] Microsoft calls for 64-bit driver support. InfoWorld Magazine Online, May 2004.
- [4] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, October 1996.
- [5] B. Bershad, S. Savage, P. Pardyak, E. Gün Sirer, M. Fitzcynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, December 1995.
- [6] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.
- [7] D. Engler, Kaashoek F, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.
- [8] A. Whitaker, R. Cox, M. Shaw, and S. Gribble. Constructing services with interposable virtual hardware. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, pages 169–182, March 2004.
- [9] R. Baron, R. Rashid, E. Siegel, A. Tevanian, and M. Young. Mach-1: An Operating Environment for Large-Scale Multiprocessor Applications. *IEEE Software*, 2(4).
- [10] D. Hildebrand. An Architectural Overview of QNX. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126. USENIX Assoc., 1992.
- [11] F. Arnaud. Give a process to your drivers. In *Proceedings of the EurOpen Autumn 1991 Conference*, Budapest, 1991.
- [12] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 181–194, December 2002.
- [13] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, October 2003.
- [14] A. Brown and D. Patterson. Embracing failure: A case for Recovery-Oriented Computing (ROC). In *Proceedings of the 2001 High Performance Transaction Processing Symposium, Asilomar, CA*, October 2001.
- [15] D. Oppenheimer, A. Brown, J. Beck, D. Hettner, J. Kuroda, N. Treuhaft, D.A. Patterson, and K. Yelick. Roc-1: Hardware support for recovery-oriented computing. In *IEEE Transactions on Computers*, vol. 51, no. 2, February 2002.
- [16] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, and H. Wang. SoftSDV: A pre-silicon software development environment for the IA-64 architecture. *Intel Technology Journal*, 3(Q4):14, November 1999.
- [17] Intel Corp. Lagrande technology architectural overview, September 2003. Order number 252491-001, http://www.intel.com/technology/security/downloads/LT_Arch.Overview.pdf.
- [18] Introduction to UDI version 1.0. Project UDI, 1999. Technical white paper, <http://www.projectudi.org/>.
- [19] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 137–151, October 1996.
- [20] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, December 1995.
- [21] K. T. Van Maren. The Fluke device driver framework. Master's thesis. University of Utah, December 1999.
- [22] C. Helmuth. Generische portierung von linux-getreibern auf die drops-architektur, July 2001. Diploma Thesis, Technical University of Dresden.
- [23] R. Minnich, J. Hendricks, and D. Webster. The Linux BIOS. In *Proceedings of the 4th Annual Linux Showcase and Conference*, October 2000.
- [24] I. Pratt and K. Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM-01)*, pages 67–76, April 2001.
- [25] Intelligent I/O (I₂O) architecture specification, Revision 2.0, 1999. I₂O Special Interest Group.
- [26] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J.N. Matthews. Xen and the art of repeated research. In *Proceedings of the Usenix annual technical conference, Freenix track*, July 2004.
- [27] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.
- [28] PCI Express base specification 1.0a. PCI-SIG, 2002.
- [29] J. Katcher. PostMark: A new file system benchmark. Technical Report 3022, Network Appliance, October 1997.

Facilitating the Development of Soft Devices

Andrew Warfield, Steven Hand, Keir Fraser and Tim Deegan

University of Cambridge Computer Laboratory, J J Thomson Avenue, Cambridge, UK

{firstname.lastname}@cl.cam.ac.uk

1 Introduction

Device-level interfaces in operating systems present a very useful cut-point for researchers to experiment with new ideas. By virtualizing these interfaces, developers can create *soft devices*, which are used in the same way as normal hardware devices, but provide extra functionality in software. Recent years have shown this approach to be of considerable interest: a few examples of block device extension include the addition of intrusion detection systems to disk interfaces [1], the development of “semantically smart” disks [2], and that of time-travel block devices [3]. Other devices, such as network interfaces, have similarly been extended.

Working at the device interface allows an examination of the functional separation between hardware and software: researchers can simulate new features as if they were properties of the device itself. As simple examples, block or network device interfaces might be extended to compress or encrypt data before it is written to disk or transmitted. Alternatively, it may be desirable to prototype entirely new devices in software, bound to existing interfaces, for instance a content-addressable disk.

Unfortunately, researchers face a challenge in extending devices in this manner. Implementors must typically modify an existing operating system to add the new functionality, often by creating OS-specific pseudo-devices. This requirement means learning the OS source and writing scaffolding code to intercept events. Moreover, where new functionality must be developed in-kernel it is difficult to debug and crashes are not contained. Finally, these low-level developments are difficult to share and maintain across systems, as they will be specific to the OS, or even specific version thereof, that it has been developed within.

This paper presents a solution to the problems associated with developing soft devices by extending the existing device interface in Xen [4]. Xen is a virtual machine monitor (VMM) for the IA32 architecture that *paravirtualizes* hard-

ware: Rather than attempting to present a fully virtualized hardware interface to each OS in a Xen environment, guest OSES are modified to use a simple, narrow and idealized view of hardware. Soft devices take advantage of these narrow interface to capture and transform block requests, network packets, and USB messages.

As an initial example of this approach, we have implemented a *block tap*, which is an interface to facilitate the development of soft devices for block device access. The block tap allows soft devices to be constructed as user-space applications in an entirely isolated virtual machine. This strong isolation from the remainder of the system allows a single soft device to work with any OS and hardware available on Xen, and allows developers to work with high-level languages and debuggers. While our approach aims to facilitate development it still provides a high level of performance, sustaining 50MB/s read throughput for disk requests in our experiments.

2 Device Access in Xen

The existing approach to device access in Xen makes use of *split device drivers*, and has been described in detail in [5]. Figure 1 illustrates a split driver: A device driver VM is granted specific access to the physical hardware that it will manage. This VM runs an existing, unmodified (e.g. Linux) device driver to access the device. In addition it runs a *back-end* driver, which provides a simple narrow interface to the device. Operating systems wishing to access the device will use a *front-end* driver, and interact with the back-end over a device channel, which is a shared-memory communications primitive.

This approach aims to improve system stability while still supporting existing device drivers by isolating drivers in a single VM, away from both other OSES and the VMM. Perhaps more importantly though, the split driver interface simplifies device access for operating systems above Xen, as an OS need only implement a single front-end driver to

Split Device Drivers in Xen

Physical driver runs in an isolated VM, connected over a shared memory device channel to a guest VM accessing the device.

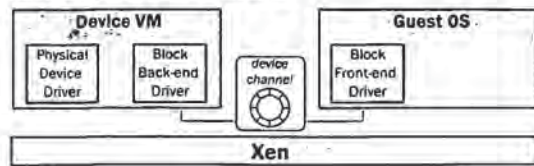


Figure 1: Split device drivers in Xen

support an entire class of devices (e.g. block storage).

Xen's current split block device illustrates exactly how narrow this interface is. The device channel shown in Figure 1 is a single page of memory shared between the two VMs. A bi-directional ring buffer is used to pass messages. Pages of data are attached and mapped separately. The interface has only three commands: READ, WRITE, and PROBE. READ and WRITE provide block-level access to data, while PROBE returns a list of accessible block devices.

3 Implementing a soft device Interface

This section describes how we have extended Xen's existing split device interface to support the development of soft devices. We describe an implementation of a block tap that allows the construction block soft devices. In designing and implementing the block tap, we have attempted to meet three general requirements:

1. **Make device implementation easy.** Our principal goal is to facilitate the development and exploration of new functionality for device interfaces. We have chosen to give developers the option of receiving device requests through a user-level interface in the soft device domain, allowing development in a safe environment with a variety of languages and tools.
2. **Do not modify the existing guest OS or device VM.** While Xen itself achieves performance by modifying the guest operating system, we desire that the soft device interface leave the attached front-end and back-end VMs unmodified. This approach allows the development of soft devices between any hardware and OS combination supported by Xen without necessitating the modification (or even understanding) of that code. Additionally, the soft device interface may be used to trace, debug, or modify an existing split device connection. As such, all soft device implementation exists within an isolated VM using only the shared-memory device channels as an interface.
3. **Maintain satisfactory performance.** We hope to use soft devices in practical situations, and not just as a

Block Tap Device Structure

The application accesses the block device interface to generate and receive block messages. The message switch has a variety of forwarding modes.

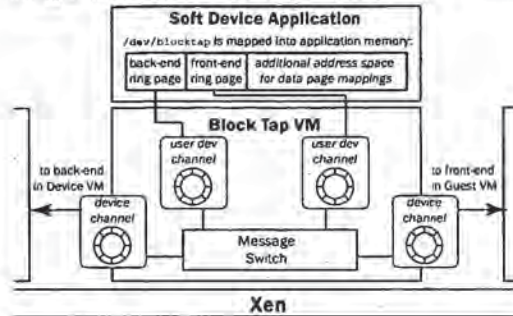


Figure 2: block tap structure.

prototyping tool. By taking advantage of the performance allowed through request batching, we hope to maintain a high level of throughput for soft devices.

The remainder of this section describes our block tap implementation addressing these requirements. The block tap is currently about 1300 lines of commented C code, and runs in a Linux-based VM.

3.1 A Switch for Block Requests

The potential ways in which a soft device interface might be used are varied. Developers may desire to simply trace request traffic in order to monitor usage patterns, they may wish to modify in-flight requests, or they may desire to construct a terminating device, which does not forward requests at all.

In order to accommodate these varied modes of operation, we have implemented the soft device interface as a request switch. The driver is plumbed into the device channel between the front-end and back-end domains. In this position, all block requests pass through it.

The new driver acts as a switch, shown in Figure 2, forwarding messages across four rings. Two of these rings are inter-VM shared memory rings as described above. They connect to the front-end and back-end drivers that the soft device interface has been placed between. Two additional rings extend from the block tap up to application space in the same VM. These rings are accessed through a character device that can be mapped by applications.

An `ioctl()` to this character device is used to set the switching mode used for block messages. Three common modes are described here, and shown in Figure 3.

`MODE_PASSTHROUGH` is the lowest-overhead switching configuration. In this mode, messages are passed straight through the driver on to the opposite ring, and completely

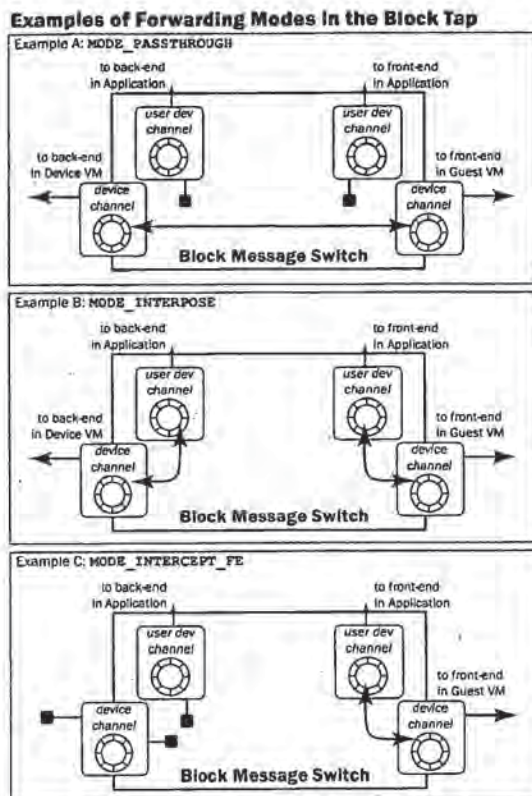


Figure 3: Examples of forwarding modes.

bypass the user rings. Passthrough can be used to implement kernel-level monitoring of block requests, or to implement soft devices in-kernel for improved performance.

MODE_INTERPOSE routes all requests and replies across the user rings. An application must attach to the block tap interface and pass messages across the two rings, allowing complete monitoring and modification of the request stream at the application level. This mode can be used to modify in-flight requests, for instance to build a compressed or encrypted block store.

MODE_INTERCEPT_FE uses only the front-end rings on the driver, disabling the back-end altogether. This mode allows the construction of full, application-level soft devices, using existing OS interfaces (such as memory, or mounted file systems) as a backing store. This mode can be used to easily prototype new functionality, or to forward block requests to a block device back-end on another physical host (after an OS migration, for instance¹).

¹OS migration is feature that we have recently added to Xen, allowing a running OS to move from one physical host to another while executing. One problem which managing migration is that local disks will be left behind.

3.2 The Application Interface

As shown in Figure 2, the user rings are exported to a character device, which is mapped by a library allowing access to the message rings and in-flight requests. Our current implementation allows chains of plugins to be attached to handle block requests. We presently have plugins to provide both copy-on-write and encrypted disks and to allow direct access to image files and remote GNBD disks.

4 Evaluation

Figure 4 shows an analysis of the impact of soft devices on block request performance with respect to both throughput and latency. Tests were performed on a Compaq Proliant DL360, which is a dual Pentium III 733MHz machine with 72.8GB Ultra3 SCSI disks.

Throughput measurements aimed to test the maximum achievable read and write speeds to the local disk. The left graph in Figure 4 shows read and write throughput moving four gigabytes of sequential data to and from disk. The three bars in the graph compare the throughput without using the block tap, using the block tap in MODE_PASSTHROUGH, and finally in MODE_INTERPOSE. As shown, our soft device interface results in a minimal degradation of throughput. We are capable of achieving 50MB/s read throughput, identical to that achieved by Xen's existing split drivers. On writes, we see about a 15% overhead; we are still investigating the source of this loss of performance.

Latency measures the per-request overhead of synchronous requests to disk. Given that disk requests are heavily batched in general, this is a less meaningful measurement for normal workloads. However, it does represent a worst-case overhead and also gives a clearer illustration of the costs that our implementation imposes. The right graph in Figure 4 shows mean request times across 100,000 4-byte synchronous writes. We see a small overhead in passing requests through the kernel of the virtual device domain in MODE_PASSTHROUGH, reflecting the cost of an additional VM context switch and request/response copy² in each direction. MODE_INTERPOSE is considerably more expensive as it adds two additional context switches and two message copies, in order to pass messages through a user-space application. There are additional costs in mapping attached data pages to user space. However, even this overhead has insignificant impact given the length of average disk seek times. We intend to explore the more demanding performance requirements of network devices in the coming months.

²Note that only the request and response structs (respectively 60 and 7 bytes) are copied on the shared memory rings. Pages of data are referenced and mapped separately.

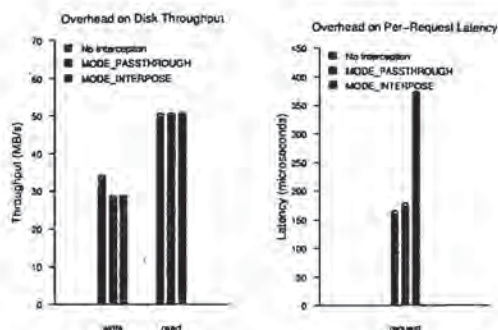


Figure 4: Overhead of virtual block devices.

5 Related and Future Work

The use of virtual machines to provide device extensibility has been explored previously in μ Denali [6]. While our intentions of facilitating development of soft devices are identical, our system is very different. μ Denali is a VMM hosting BSD VMs, which access devices over micro-kernel-style IPC through Mach's port abstractions. Xen does not use synchronous micro-kernel-style IPC; instead it detaches message transfer (copying/mapping pages between VMs) from notification (virtual interrupts), to achieve high throughput through message batching. We feel that these differences are interesting for two reasons: first, the design presented here represents a considerably different approach to that presented in [6]. Second, as Xen is a publicly available VMM and supports the production use of several popular OSes, we hope that the availability of this work is of general interest to other researchers.

The block tap itself is similar in some ways to the FreeBSD GEOM and Linux EVMS projects, both of which provide a great deal of extensibility to their respective systems' block device interfaces. The block tap aims to provide similar extensibility, but in a virtualized environment, thus catering to a range of operating systems. The use of virtualization also allows strong isolation for soft devices, enhancing stability.

We view the soft device interface as an enabling tool for future work. As mentioned earlier, other projects have explored the use of interposition on device interfaces for such goals as intrusion detection [1], semantically smart disks [2], and fault diagnosis [3]. Two projects that we are particularly interested in exploring over these interfaces in the coming months are described here.

Parallax - a cluster storage system for virtual hosts. Managed virtual machines present different file system requirements than are currently available through distributed or cluster file systems, or network-attached storage. We are

currently building a cluster-based storage system to provide high-availability storage for virtual OSes that may migrate between physical hosts, return to arbitrary historical snapshots, and which likely exhibit a large amount of commonality in terms of their system images.

Data-stream watchpoints for pervasive debugging. Interposing on block and network requests will allow existing OS debugger work above Xen to be extended to enable break and watch points on specific content. We intend to allow debugging to be triggered if a specific file is modified, or certain network traffic is sent or received.

6 Conclusion

This paper has briefly presented the implementation of a virtual device interface for OSes on the Xen VMM. Our implementation enables the construction of new functionality to trace, interpose on, or extend existing block device interfaces. We look forward to extending this approach to support network and USB devices, and to building systems above it.

References

- [1] A. Pennington, J. Strunk, J. Griffin, C. Soules, G. Goodson, and G. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *Proceedings of the USENIX Security Symposium*, August 2003.
- [2] M. Sivathanu, V. Prabhakaran, F. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, March 2003.
- [3] A. Whitaker, R. S. Cox, and S. D. Gribble. System administration as search: Finding the needle in the haystack. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI '04)*, December 2004.
- [4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 164–177. ACM Press, 2003.
- [5] K. Fraser, S. Hand, I. Pratt, A. Warfield, R. Neugebauer, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the First Workshop on Operating System and Architectural Support for the on-demand IT Infrastructure (OASIS-2004)*, October 2004.
- [6] A. Whitaker, R. S. Cox, M. Shaw, and S. D. Gribble. Constructing services with interposable virtual hardware. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

Hierarchical Disk Sharing for Multimedia Systems

Joel C. Wu
jwu@cs.ucsc.edu

Scott Banachowski
sbanacho@cs.ucsc.edu

Scott A. Brandt
sbrandt@cs.ucsc.edu

Computer Science Department
University of California, Santa Cruz
Santa Cruz, CA 95064

ABSTRACT

Systems that use or serve multimedia data require timely access to data on hard drives. To ensure adequate performance users must either prevent overload of disk resources, or use real-time algorithms that rely on intricate knowledge of disk internals to meet deadline requirements. We have developed Hierarchical Disk Sharing (HDS) to allow disks to be fully utilized while sustaining a bandwidth reservation, without requiring detailed knowledge of the drive internals. HDS uses a hierarchy of token bucket filters to isolate disk access among clients and groups of clients, and to allow for reclaiming of unused bandwidth. We discuss the design of HDS and present our implementation in a Linux block device driver, demonstrating the effectiveness (and limitations) of this approach.

Categories and Subject Descriptors

D.4.2 [Operating Systems]: Storage Management—Secondary storage; D.4.8 [Operating Systems]: Performance

General Terms

Algorithms, Design, Performance

Keywords

Storage, Disk, QoS, Multimedia, Soft Real-Time

1. INTRODUCTION

Multimedia computing is ubiquitous and it is common to find systems concurrently supporting both multimedia and general-purpose workloads. This is also true for storage, as multimedia files reside on the same storage devices as other data. To meet their timing constraints, multimedia applications must access stored data in a timely manner or experience undesired performance such as jitter, drop-outs, or other audio and video glitches.

Applications typically access data without regard for others and may successively request multiple blocks as fast as

the storage device permits. This may negatively impact performance when multimedia applications, which require sustained disk bandwidth to read and decode data on time, must compete for disk resources while other applications use it for non-real-time purposes. Since we cannot in general expect applications to cooperatively share the disk, it is the role of the operating system to provide resource isolation by allowing time-critical applications to access data at reserved levels of service (*reservations*), or by limiting the amount that competing applications may interfere with each other (*fairness*).

This paper describes Hierarchical Disk Sharing (HDS), an algorithm for reserving and partitioning the bandwidth to disk and other block storage devices among different applications, or groups of applications (which we call *classes*). The principle behind HDS is based on the hierarchical link sharing for networks [6]. HDS provides capability that is absent in current commodity operating systems. One of our main design goals is that the reservation mechanism be independent of high-level features like file-systems, and low-level features like disk schedulers, so that it can be employed across many systems, including storage network devices. Therefore we chose to implement our prototype of HDS in the block device layer of the Linux kernel.

2. BACKGROUND AND RELATED WORK

Time-sensitive storage access is addressed by real-time disk scheduling algorithms [7]. Generally, real-time disk schedulers are designed for homogeneous applications such as video-on-demand. In such systems, disk requests have deadlines. The requests are typically ordered based on some combination of real-time scheduling technique and disk seek optimization scheme (e.g. SCAN-EDF [8]). These algorithms do not support mixed-workload environment well.

More elaborate disk schedulers (whether real-time or not), which take into account the seek time and rotational latency, require the use of accurate disk models. Using such low-level information may provide predictable service guarantees [9]. A challenge facing the practical deployment of these schedulers in commodity systems lies in their dependency on intricate knowledge of drive internals (which must be determined by probing the disks [5, 13]). This run-time information is difficult to obtain for an external disk scheduler, and low-level disk properties must be known and well-characterized for each different model of drive. Exacerbating the issue is the increasing intelligence of drive firmware, which limits the effectiveness of these schedulers to special-purpose systems. Our goal is to provide better support using com-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

NOSSDAV'05, June 13–14, 2005, Stevenson, Washington, USA.

Copyright 2005 ACM 1-58113-987-X/05/0006 ...\$5.00.

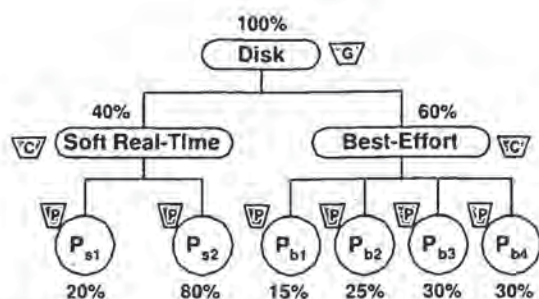


Figure 1: HDS allows arbitrary mix of shares controlled by Global bucket, Class buckets, and Process buckets.

modity equipment and enable shaping of bandwidth without relying on the disk scheduler.

Real-time disk schedulers provide fine-grained control of disk requests based on the deadlines of the disk requests. For many applications, this amount of control is unnecessary. For example, for many soft real-time applications it is more important to achieve a constant data rate (a constant quality of service (QoS)) than to meet the deadlines of individual disk requests. Our approach is to provide better QoS over aggregate data rates.

Some QoS-aware schedulers are designed specifically to address mixed workloads [4, 11, 12]. These are typically two-level schedulers that classify disk requests into categories such as real-time, best-effort, and interactive. Each class has its own scheduler, and the requests from different classes are merged by a meta-scheduler. The number of classes is limited and static. Our multi-level approach allows the dynamic creation and removal of classes at run-time. YFQ [1] offers similar capability to HDS, using different mechanisms.

The techniques we employ in HDS are already used for managing networking resources. Our resource allocation policies are based on the hierarchical link-sharing mechanism for networks [6]. Like that system, we isolate traffic by separating the requests of different applications or types of application; disk blocks are analogous to packets, and bandwidth represents the number of blocks accessed over a period of time. The mechanism we use to control disk bandwidth is also based on a networking technique: token bucket filters [10], which were designed for network flow control. However, unlike network bandwidth, the available disk bandwidth fluctuates due to variations in seek times for different disk requests, complicating its management.

3. HIERARCHICAL DISK SHARING

Traditional disk access is best-effort, and there are no timing guarantees. Acceptable performance is achieved when the disk is not overloaded. When demand for disk bandwidth exceeds the supply, all of its users may experience performance degradation.

Our approach to this problem is to provide a mechanism that allows reservations of disk bandwidth, graceful degradation under heavy load, and reclaiming of unused reservations. A hierarchical structure for resource sharing provides a basis for meeting all of these goals. This section describes the Hierarchical Disk Sharing (HDS) approach.

3.1 Disk usage accounting

Specifying a disk reservation by bandwidth is intuitive, but disk bandwidth is not constant; service times vary depending upon the initial position of the read-write head, the position of the requested data on the disk, the low-level disk scheduling algorithm, etc. Translating a bandwidth requirement into low-level disk operations is a complicated task [2]. Alternatively, specifying a disk reservation by a reserved time-slice (instead of bandwidth) may result in different amounts of data being retrieved per time-slice, and the specification of disk time is not intuitive from the user's perspective.

Existing reservation-capable schedulers must contend with this issue. The Cello [11] scheduler presents two methods of accounting, either by size or time. Some schedulers allow reservations in terms of number of requests [1, 12]. However, requests may also vary in size and service time.

To fulfill QoS goals, the system must provide performance in line with the users' expectations, regardless of the amount of work that the disk is actually doing on behalf of different users. We expect users to perceive the quality of service for disk by the bandwidth (data rate) that it can provide. Therefore HDS accounts for disk usage in terms of bandwidth and does its accounting based on the actual amount of data transferred.

3.2 Hierarchical structure

In HDS, a disk's bandwidth is divided between applications in a hierarchical tree structure; an example is pictured in Figure 1. Each leaf node represents a point of control for accessing the disk. Therefore, each leaf-node is associated with an individual client of the disk; in our implementation a leaf node corresponds to a Linux process. When a process first attempts to access the disk, a leaf node is created and added to the tree. When it quits, its node is removed.

Non-leaf nodes are called *classes*, and represent a group of clients. The children of a class node may be leaf nodes or other class nodes. For example, a class node might represent a user logged onto the system, with all its children representing processes created by this user. The parent of the user class node may itself be a class node representing the user's department. Figure 1 demonstrates using classes to isolate best-effort and real-time processing, an approach useful for multimedia servers where the requests with time constraints should be isolated from other traffic.

Hierarchies may be created to control disk accesses in many flexible configurations, and we expect that a system administrator or other expert may want to choose how to set up a particular system. Our system has an interface for constructing the desired class hierarchy, including dynamically adding and deleting classes; for brevity the interface description has been left out of this paper.

3.3 Hierarchical reservation

Each node x has an associated reservation r_x , determined by the reservations of nodes above it in the tree structure. There are two modes for a node to specify its reservation: either an *absolute* fraction f_x , or a *relative* fraction, based on a weight w_x , of the parent node's reservation. The root node has an absolute fraction of 1.

If a node x has an absolute reservation f_x (between 0 and 1), its reservation is this fraction of its parent's reservation. For example, because the root node has $r_0 = 1$, a

child of the root with $f_x = 0.4$ will have a reservation of $r_x = f_x r_0 = 0.4$. The sum of absolute fractions among any node's children may not exceed 1.

A class's bandwidth that is not used by absolute reservations is shared by its other children in proportion to their relative weights. For example, consider a class node (node p) with reservation $r_p = 0.5$, and three children nodes (a, b, and c), the first with absolute fraction $f_a = 0.4$, and the others with relative weights $w_b = 4$ and $w_c = 6$. Because node a is absolute $r_a = 0.4r_p = 0.2$, leaving $0.6r_p = 0.3$ left for nodes b and c. Node b's reservation is $r_b = (0.6r_p)4/(4+6) = 0.24r_p = 0.12$, and node c's reservation is $r_c = (0.6r_p)6/(4+6) = 0.36r_p = 0.18$.

In the default setting, clients are added to a parent node with equal weight to promote fair sharing. When a client needs a higher level of service than others, it may do so by either increasing its weight or requesting an absolute fraction of its parent bandwidth. If the nodes on the path from a client to the root all have absolute reservations, then the client effectively reserves a static fraction of the total disk bandwidth; if any node in this path has a relative reservation, the node's reservation may vary when other nodes join or leave the structure. HDS allows administrators to set permissions for adding classes or nodes, changing reservations, admission control, etc. These policy-based controls are outside the scope of this paper.

3.4 Token bucket implementation

In the previous section we explained that each node in the hierarchical tree structure has a value representing its current reservation of the disk. We now describe our mechanism for enforcing these reservations using token bucket filters at each node of the hierarchy. Current Linux distributions include a Hierarchical Token Bucket (HTB) filter for the network driver [3]. We developed HDS for the block device driver independently of the network driver—although the basic goals are essentially the same, they share no common code and the algorithms differ.

Disk bandwidth may be controlled at different points in the I/O stack. HDS is at the block-device layer, which is below the file system and above the disk scheduler. The regulation of disk bandwidth in HDS is implemented using token bucket filters. In order to make disk requests, a client must possess tokens. In HDS, each token represents 1 KB of data, meaning a request for 16 KB of data requires 16 tokens. Each node x in the hierarchy has an associated bucket, which may hold up to N_x tokens. When a client request is serviced, tokens are removed from its bucket. Tokens are replenished at a rate corresponding to the client's reservation. If the root token rate is T_0 , then its child node x with reservation r_x will replenish tokens at rate $T_x = r_x T_0$. The root token rate represents the entire bandwidth of a disk. For example, if a disk supports an average throughput of 20 MB/second, the root token rate $T_0 = 20K$ tokens/second.

Although every node has a token bucket, only leaf nodes make requests. The token buckets of non-root nodes facilitate sharing of bandwidth. In addition to its own tokens, a node may use tokens from its parent (which in turn may use those of its parent). The effect is that unused bandwidth is shared first among nodes of the same class, then among parent class, and eventually, globally.

```

replenish(x)
   $c_x = \max(c_x + (t - l_x)T_x, N_x)$ 
   $l_x = t$ 

decrement(x, size)
   $c_x = c_x - \text{size}$ 
  if( $P_x \neq \text{root}$ ) decrement( $P_x, \text{size}$ )

node can_make_request(x, size)
  if( $\text{size} > c_x$ ) replenish(x)
  if( $c_x \geq \text{size}$ ) return x
  if( $P_x \neq \text{root}$ ) return can_make_request( $P_x, \text{size}$ )
  return null

make_request(x, size)
   $y = \text{can\_make\_request}(x, \text{size})$ 
  if(y)
    decrement(y, size)
  else
    decrement(x, size)
    sleepfor(( $\text{size} - c_x$ )/ $T_x$ )
  endif
  do disk access

```

Figure 2: Functions used by the HDS algorithm.

Definitions. The following definitions are used to describe the algorithm: The current time is t . For a node x , P_x is its parent, N_x is its bucket size, c_x is the number of tokens currently in its bucket, T_x is the token rate, and l_x is the last time its bucket was replenished. The pseudo-code for the algorithm is in Figure 2.

To make a disk request, the client calls the function `make_request()`, with arguments indicating its node and the size of the request (all client nodes are leaf nodes). If the client, or any of its ancestors, has enough tokens to satisfy the request (determined by `can_make_request()`), then the disk access proceeds immediately. Otherwise, the calling process is suspended until enough time passes to acquire the necessary tokens. This blocking mechanism forces clients to throttle their request rate to their token rate, thus enforcing the bandwidth reservation.

The function `can_make_request()` determines if a node has enough tokens to meet the request. If the node's current token supply is low, it calculates the number of tokens that should be replenished since its last request. If the node still does not have enough tokens to service the request, it determines if a higher-level node in its hierarchy has enough by recursively calling `can_make_request()` on its parent. If no node in the caller's path to the root may satisfy the request, `can_make_request()` returns `null`, otherwise it returns the identity of the node that satisfies the request.

Before servicing the disk request, `make_request()` decrements the tokens from the bucket of the node supplying the tokens. Decrement is a recursive function, so tokens are also removed from the buckets of all higher-level nodes.

3.5 Properties

Only leaf nodes initiate requests. Because every node has its own supply of tokens, HDS guarantees that every node is always able to freely request up to its reservation in disk requests (allocated at rate T_x , and corresponding to x 's reservation r_x). HDS also allows clients to consume any extra bandwidth by using tokens supplied by parent nodes

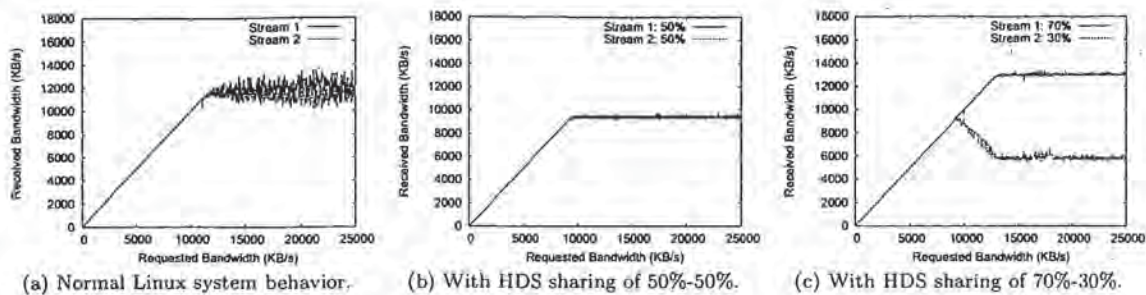


Figure 3: The effect of overload on throughput.

recursively, up to the root node. This hierarchical structure provides a method for clients to reclaim bandwidth that is reserved, but not used, by others.

When a node drains tokens from its own bucket, it also drains those from higher-order buckets. The result is that when a class's children make disk requests, the class's tokens will be drained as well. If some of the children are not fully using their reservation, the parent will have surplus tokens. These tokens are available to other children when their own supply runs out, so that a node that has exceeded its reservation may still be able to proceed. Bandwidth isolation is preserved not only between leaf nodes, but at the class level and, in fact, at every level of the hierarchy.

4. EXPERIMENTS WITH HDS

We ran several experiments to demonstrate the ability of HDS to shape disk traffic, using synthetic applications to generate disk workload. We focus mostly on read workloads both because multimedia is typically read intensive and because write performance is often aided by buffering. Our test system is a 1.5 GHz P4 with 512MB of RAM. The disk is a Seagate ST340810A IDE drive formatted with the ext2 file system.

Figure 3 shows a situation where disk bandwidth has become saturated. Two processes are reading from the disk simultaneously, at the same rate. The x-axis shows the requested bandwidth and the y-axis shows the measured received bandwidth. Figure 3(a) shows the result on unmodified Linux. Both processes receive their desired bandwidth until the disk became saturated with requests (i.e. the sum of requested bandwidth exceeds the total disk bandwidth). There is no isolation, so at that point actual throughput is unpredictable and varies considerably.

HDS provides reservation and isolation of bandwidth. Figure 3(b) shows the same experiment with HDS, where each task reserves equal relative weight. At saturation the bandwidth divides evenly between the streams and the achieved throughput is very stable. This fair-sharing comes at the expense of slightly lower overall disk throughput because we limit the number of requests. Section 5 discusses future work for alleviating this effect by varying the root token rate based on observed performance.

Figure 3(c) shows the same workload again, while demonstrating HDS's reservation capability. We allocated 70% of the disk to stream 1 and 30% to stream 2. Below disk overload, each process receives what they request. As the requested bandwidths exceed the disk capability, the reservation mechanism begins to take effect. Stream 1 continues to receive a higher level of service based on its share. Once the 70%-30% share is reached, the bandwidth received by each process remains constant as the rate of disk requests increases. The average throughput for Stream 1 at full rate is 12.7 MB/s, about 70% of the total disk bandwidth of 18 MB/s, and for Stream 2 it is 5.7 MB/s, about 30% of the total bandwidth.

The next experiment shows the ability of HDS to provide hierarchical resource sharing. We created two classes, A and B, each reserving 50% of the disk. Stream 1 belongs to Class A, so it reserves 100% of the class reservation. Streams 2 and 3 belong to Class B, and reserve 65% and 35% of Class B's reservation, respectively. Figure 4 shows that all three streams receive bandwidth corresponding to their allocation, with no interference from each other.

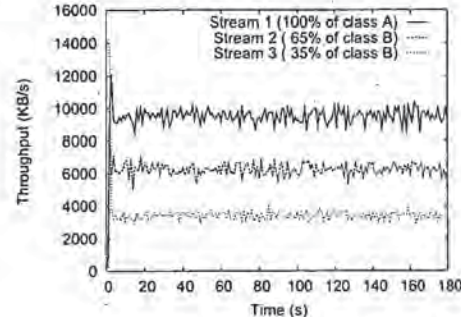


Figure 4: Isolation of bandwidths. Class A and B reserve 50% each. All streams are greedy.

Excess bandwidth may be available when a process needs more than its reserved share. Figure 5 shows this scenario. In this experiment, there are two classes and three streams. Class A and B each reserve 50%. Stream 1 and 2 belong to Class A and reserve 80% and 20% of its bandwidth. Stream 3 belongs to class B, so receives 100% of its bandwidth. At the beginning, only Stream 1 is active. Although its total share is only a fraction of the total bandwidth (its share is 40%), because no other tasks are active it receives the total disk bandwidth. At time 60 Stream 2 becomes active. There is still excess bandwidth because Class A's share is only 50%. The excess bandwidth is distributed to Streams 1 and 2. From time 60 to time 120, they receive

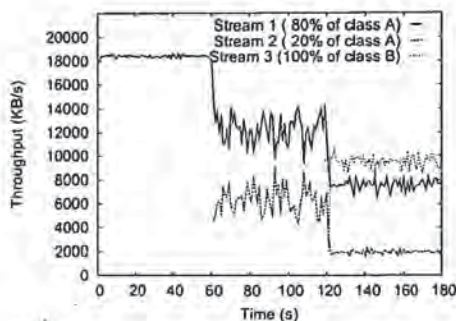


Figure 5: Using unassigned bandwidth. Class A and B reserve 50% each. All streams are greedy.

their fair-share plus the excess bandwidth. Excess bandwidth is allocated on first-come first-serve basis, accounting for the observed variation in actual rate (this variation is a topic for future investigation). At time 120 Stream 3 begins and, now fully loaded, the nominal reservations are enforced.

5. FUTURE WORK

The throughput of a disk during overload depends on the workload. Since the totality of resource may vary, proportional sharing is often used to allocate a fraction of the bandwidth to different shares [11, 1]. However, because the total disk bandwidth varies, these fractional guarantees are not constant. In many circumstances it would be more useful to have assurances in terms of data rates, and not in terms of a fixed fraction of a dynamically changing total bandwidth.

HDS is able to provide assurance in terms of data rates with *absolute* reservation and knowledge of the root node bandwidth. However, the unpredictable nature of storage also presents a problem for HDS. Thus far we have assumed that the disk bandwidth is fixed. Although convenient, as already noted, this assumption is not true; the throughput of a disk during overload depends on the workload. This is highlighted in the experiment shown in Figure 6 (without HDS), where we introduced 8 greedy streams in 25 second intervals. The upper line shows total throughput. As the offered load increases, the total achieved bandwidth varies significantly.

HDS uses a conservative estimate of total bandwidth. This under-utilizes the disk, but allows HDS to provide the reserved bandwidth. The more aggressively HDS utilizes the disk, the less effective it is at providing isolation and sharing. In Figure 7 (which has the same workload as Figure 4), we show what happens as we increase the bandwidth of the root node by 384 KB/s every 20 seconds. In this case, the share mechanism becomes less effective, failing completely about 2 minutes into the experiment, after a 2 MB/s increase in disk bandwidth.

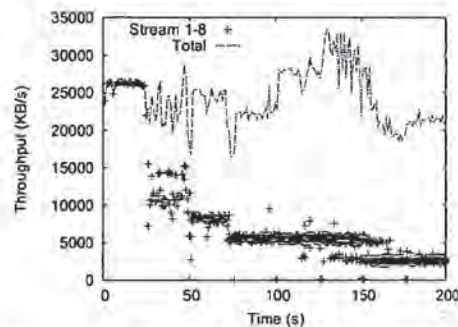


Figure 6: Effect of changing workload on total throughput.

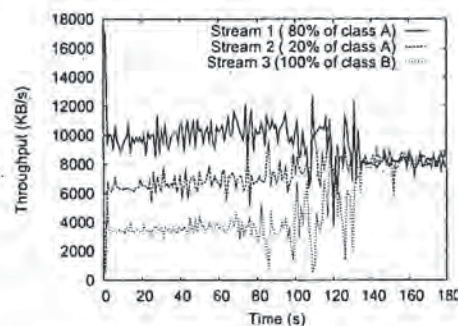


Figure 7: Effect of increasing root node bandwidth on sharing.

We plan to increase the performance of HDS by addressing this issue. One possible solution is to adapt the total rate of the root node based on the current disk performance or the effectiveness of the current reservations. For example, we may monitor the actual rates at nodes during overload and, if they do not match their reservations, we may adjust the root token rate downward until they do.

We implemented a heuristic to show the benefit of adjusting the root node's bandwidth in response to performance, shown in Figure 8. Stream 2 begins at time 40, and the two streams maintain their relative share of 3:1 from time 40 to 70. The adaptive mechanism is triggered at time 70; it attempts to increase the total bandwidth without violating the reservations. The algorithm increases the global token rate if the current measured shares are accurate; otherwise it decreases the global token rate. After the adjustment at time 70, the throughput of Stream 1 is increased while Stream 2 maintains its constant rate. The algorithm is slow to adapt, but demonstrates the potential of this approach.

With a dynamically changing bandwidth for the root node, the *absolute* and *relative* weighted sharing of HDS can be augmented by a third type of reservation, *absolute bandwidth* reservation, where a portion of the bandwidth is set aside for reservation in terms of bandwidth, and the rest of the bandwidth is available for proportional reservation by either *absolute* or *relative* weighted sharing. The portion of bandwidth set aside for absolute reservation can be pro-

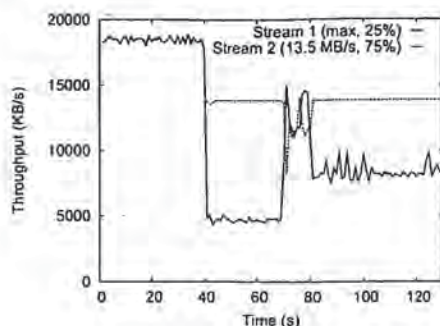


Figure 8: Maximizing total throughput.

tected during overload by monitoring and throttling the rate of the proportional share portion.

Adapting the root token rate dynamically also solves another problem for HDS: the throughput of a disk is unknown without testing. In our experiments, the default token rate was set to a value we predetermined based on previous average throughput measurements. If deployed in a commodity system, the disk performance will be unknown. Using an adaptive root token rate will allow HDS to dynamically discover appropriate rates when applied to an unknown disk.

Other future work includes investigating the interaction between HDS and low-level disk schedulers. Although one of our stated goals is to avoid dependence on the disk scheduler, our algorithm need not ignore that many low-level schedulers have similar goals which we can leverage by managing the ordering of related requests. Another area to investigate is the effect of prefetching on our token-cost structure. In addition to the streaming workloads of file data, the block device driver where we implemented HDS sees requests for metadata and prefetches from the file system, facts that HDS has so far ignored.

6. CONCLUSION

Hierarchical Disk Sharing (HDS) supports flexible and dynamic sharing of storage resources. It allows processes to reserve disk bandwidth and effectively isolates disk clients, providing them with their reserved share. HDS uses a hierarchical structure for allocating disk bandwidth that simplifies reservations and allows processes to reclaim unreserved or reserved but unused bandwidth. We implemented HDS in Linux and demonstrated its ability to isolate the disk traffic of clients and provide storage quality of service. During overload, clients receive at least their reserved share of bandwidth, in isolation from other competing loads. The effectiveness of HDS is determined by the choice of the algorithm's maximum disk bandwidth. We are currently developing an algorithm that will allow the entire disk bandwidth to be dynamically adjusted, automatically adapting to new platforms and changing workloads. Our eventual goal is to apply HDS in an overall end-to-end QoS framework for distributed storage.

Acknowledgments

This research was supported in part by Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratory under contract B520714; also by Intel Corporation and a DOE High-Performance Computer Science Fellowship.

7. REFERENCES

- [1] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *IEEE International Conference on Multimedia Computing and Systems*, volume 2, pages 400–405, June 1999.
- [2] S. Childs. Portable and adaptive specification of disk bandwidth quality of service. In *Proceedings of the 9th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, June 1999.
- [3] M. Devera. Hierarchical token bucket. <http://luxik.cdi.cz/~devik/qos/htb/>.
- [4] Z. Dimitrijevic and R. Rangaswami. Quality of service support for real-time storage systems. In *Proceedings of the International IPSI-2003 Conference*, October 2003.
- [5] Z. Dimitrijevic, R. Rangaswami, and E. Chang. Diskbench: User-level disk feature extraction tool. Technical report, UCSB, November 2001.
- [6] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, 1995.
- [7] J. Gemmell, H. Vin, D. Kandlur, P. Rangan, and L. Rowe. Multimedia storage servers: A tutorial and survey. *IEEE Computer*, 28(5):40–49, 1995.
- [8] A. L. Reddy and J. Wyllie. Disk scheduling in a multimedia I/O system. In *Proceedings of ACM Conference on Multimedia*, pages 225–233. ACM Press, 1993.
- [9] L. Reuther and M. Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS). In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*. IEEE, December 2003.
- [10] S. Shenker, C. Partridge, and R. Guerin. Specification of guaranteed quality of service. *RFC 2212*, September 1997.
- [11] P. Shenoy and H. Vin. Cello: A disk scheduling framework for next generation operating systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 44–55. ACM Press, 1998.
- [12] R. Wijayarathne and A. L. Reddy. Integrated QOS management for disk I/O. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 487–492, June 1999.
- [13] B. Worthington, G. Ganger, Y. Patt, and J. Wilkes. On-line extraction of SCSI disk drive parameters. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 146–156. ACM Press, 1995.

Improve your odds with Infosys Predictability



Right Questions



Business Insights



Integrated Solutions



Value Multipliers

Xen and Server Consolidation

— Srikanth Sundarrajan, Subhabrata Bhattacharya

Abstract

Enterprises are witnessing a server sprawl as low cost x86 boxes make their way into datacenters. Application provisioning on dedicated servers and provisioning hardware capacity for peak loads have led to unprecedented server proliferation and abysmally low utilization. Virtualization technology has the potential to play a significant role in enabling logical server consolidation. It enables enterprises to achieve higher utilization and manage their hardware resources better, thus reducing Total Cost of Ownership (TCO) and help contain IT spending. It is a promising technology that is seeing steady adoption. A low overhead and highly scalable virtualization solution is the key to this adoption. Xen [1] an ideal solution for server consolidation based on para-virtualization is gaining currency.



Introduction

Virtualization is a technique through which hardware resources viz. processors, storage, I/O and network on one or more machines can be transformed through hardware/software partitioning, time sharing and simulation/emulation into multiple execution environments, each of which can act as a complete system.

Computing power has doubled almost every other year as predicted by Gordon Moore. Processors based on IA32 have particularly been able to take advantage of this, resulting in a proliferation of IA32 based servers in most enterprises today. High acquisition and maintenance costs of mainframes and large UNIX machines have compelled enterprises to go with off the shelf servers based on IA32 architecture, which were cheaper by several orders of magnitude.

Enterprises have typically deployed applications on dedicated IA 32 based servers in their data centers for two reasons. Firstly, to provide isolation and secondly to accommodate peak hour utilization and sudden utilization surges. Isolation is important to ensure that applications don't interfere with each other in terms of the configuration or environment. Also, security or other faults in one application don't compromise co-hosted applications. Secondly, peak hour utilization factors have to be considered to provision the servers to guarantee and maintain Service Level Agreements (SLA) for the application. This has resulted in heavily over-provisioned server farms in the data centers.

Enterprises are now looking for ways to improve the average utilization of their servers, reduce maintenance cost and also retain the Quality of Service (QoS). Consolidating multiple applications on a single physical server can solve issues related to lower utilization, however applications won't be isolated sufficiently to address the security, quality of service, fault tolerance and application incompatibility issues. Workload management solutions can perhaps address issues related to provisioning and QoS. Virtualization is the only technology currently available that makes a logical server consolidation possible without compromising on any of the desired features.

Virtualization can partition a physical machine into multiple smaller machines of desired configuration and each capable of hosting different Operating Systems and environment, thereby creating an illusion of multiple machines, while in fact residing on single physical machine.

There are four popular approaches to system virtualization depending on layer at which the system is virtualized. These are Full System Virtualization, Para Virtualization, Hosted Virtual Machine Architecture and Shared Kernel virtualization. A high level definition of each of these approaches is presented below.

Full System Virtualization is a technique in which the target hardware is emulated in full by directly executing some instructions on the hardware and some through the VMM. The major advantage with this technique is that the guest operating system (that runs on VMM) or the applications that are executed on the guest operating system need not be modified. With the current implementation of IA 32 architecture, Full System Virtualization is not achievable. Intel's

VT [2] and AMD's Pacifica [3] will make this possible in future versions of their processors

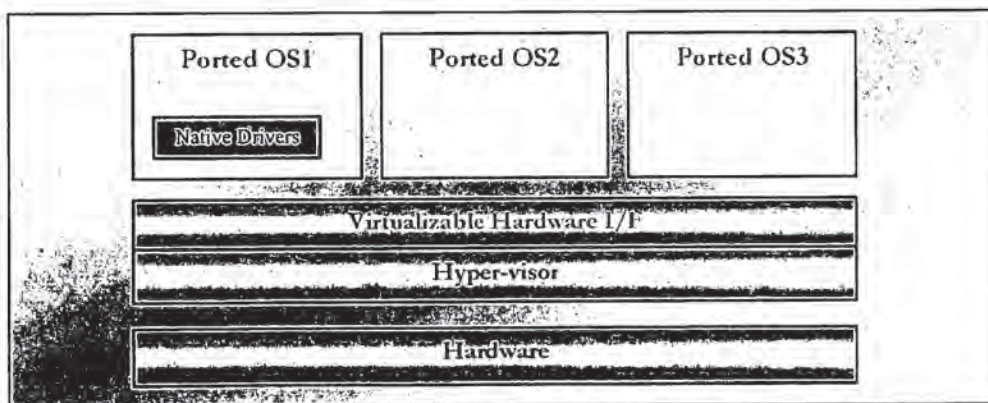
Para-Virtualization requires the operating system to be modified to avoid instructions that cannot be fully virtualized by going through the VMM / hypervisor to execute those. Both the host OS and the guest OS must be modified and recompiled for para-virtualization. However applications don't need to be modified.

Hosted Virtual Machines adopts a method wherein the VMM and guest OS run in the user space of the machine and the Host OS provides the services required by VMM. Applications running on the host OS and the guest OS share the same user space. This doesn't require any modification to the guest Operating System or the user applications; however the additional level of indirection normally causes heavy performance overheads.

Shared OS Virtualization is an approach in which a machine can be partitioned into multiple containers with isolated process, memory, devices and file system while sharing a single kernel image. Each container is called a Virtual Private Server.

Para Virtualization

Para Virtualization is a technique in which the instruction set of the hardware (which doesn't support virtualization) is modified to a fully virtualizable instruction set. Modifying the hardware instruction set means that the Operating System (read Kernel) also needs to be ported to the newer instruction set. In most cases applications don't require porting because, they don't use instructions that operate with hardware's internal registers or machine state.



Para-Virtualization Schematic Architecture

By creating a parallel virtualizable instruction set, a hypervisor can be built which can host the kernels of the guest OS. No run-time translation is required in this case, as the kernel is ported at compile time and the performance in most cases is similar to that experienced in full system virtualization.

The biggest disadvantage of this technique is that the kernel needs to be ported for the hypervisor. This may not be possible with proprietary OS/kernel such as Microsoft Windows®.

Native device drivers are installed in one of the guest OS instances and typically the other guest OS access them through device channels as in mainframes and in other cases the hypervisor may include device drivers for limited number of devices and the guest OS may access the devices through the hypervisor. Both in the first and the second case the number of context switches between the guest OS and VMM are limited, resulting in a near native I/O bandwidth

Xen

Xen is an open source virtualization solution conceived by researchers from the University of Cambridge which is based on para virtualization. The heart of Xen is the Virtual Machine Monitor termed as the hypervisor. This hypervisor exposes itself a platform for the multiple Virtual Machines on which guest operating systems run. Accessibility of hardware is completely abstracted through the hypervisor. The guest operating systems need to understand the hypervisors hardware abstraction layer in order to access the underlying hardware; hence their kernels are modified accordingly.

Xen Architecture

CPU Virtualization

Intel x86 processors have the concept of privileged modes to execute instructions. These modes can be visualized as concentric rings. The innermost ring, also known as the ring 0 is capable of executing the most privileged instructions. The privileges decrease as one moves outwards. All Intel processors have 4 levels of privileges. These modes or rings are numbered 0 to 3. Typical operating systems (without para virtualization) are executed in ring 0 and the applications running on top of these operating systems are executed in ring 3 leaving ring 1 and 2 unused.

With para virtualization as in the case of Xen, the innermost ring executes the Xen Virtual Machine Monitor. The guest operating systems execute in the subsequent ring 1. Applications running on top of these guest operating systems get the least privileged mode or ring 3. The hypervisor being the most privileged entity is capable of manipulating the virtual machines which have lesser privileges. This approach helps to ensure that the hypervisor possesses the highest privilege, while the guest OS executes in a higher privileged mode than the applications and is isolated from the applications. Privileged instructions issued by the guest OS are verified and

executed by the hypervisor.

Xen maintains a registry to map each exception it encounters with the handler that processes it. The data structure used to implement this registry is similar to those on x86 hardware used to maintain information on exception handlers. Page fault exceptions are treated with a slightly different approach. Exception handlers in the guest OS are modified to read the faulting address from its own stack instead of a privileged processor registers. System calls that do not involve privileged instructions are handled without the intervention of ring 0 routines, thereby enhancing the performance. Each operating domain has its own share of CPU cycles allocated by a Borrowed Virtual Time (BVT) scheduler. A switchover from one domain to the other should take place in near real time whenever a request arrives. Through virtual time warping in BVT scheduling, latency sensitive domains can borrow future allocated CPU time, thereby not disrupting long term CPU sharing. Since this scheduler has comparable performance with real-time schedulers and has very little overhead, it naturally becomes a choice for Xen. Other schedulers can also be integrated with the hypervisor code.

Guest domains in Xen are given the notions of real time, virtual time and wall-clock time. Real time is nanoseconds passed since the machine booted and it is totally subject to the accuracy of the processor's cycle counter. Virtual time is the time taken by a domain while it is executing and this helps the scheduler to keep account of the total amount of time a particular domain is entitled to. Wall-clock time, on the other hand, is obtained by adding an offset to the current real time. Timers for real time and virtual time are implemented in the guest operating systems.

Memory Virtualization

In order to reduce latencies involved in accessing memory locations in the physical address space, processors have caches called Translation Lookaside Buffers (TLB). x86 based processors have implemented page table walking state machines to resolve physical memory addresses. This is totally administered by the processor hardware and the operating system does not usually have any role to play here. However, with some tradeoffs with the performance TLBs can be managed by software. This mechanism is exploited in Xen to virtualize the entire physical memory. Xen divides the physical address space into several compartments to be used by the various domains. Whenever there is a switch from one address space to the other, TLB needs a flush to update the process table entries pertaining to each guest OS. With Xen code residing on the topmost 64MB section of each of the address spaces used by the guest operating systems, the Xen hypervisor doesn't need to flush every time such a switchover occurs. To ensure address space isolation, updates on the segment descriptor tables and page tables by guest OS pass through a validation by the hypervisor. This is done by a hypercall. Hypercalls are queued at the guest OS, but pending address updates are always committed before a flush takes place.

Initially each domain is statically allocated a specified amount of physical memory at the time of creation. However, if there is need for more physical memory inside the domain, then it could be extended by using a balloon driver subject to the availability of total physical memory. Memory

pages can also be released to the hypervisor from inside the domains for further allocation. Memory pages allocated by Xen need not be contiguous. Xen provides a shared translation array to map hardware address into physical address which can be read by all domains and updated after being validated by the hypervisor.

I/O Virtualization

Xen doesn't fully emulate the underlying hardware devices, contrary to the fully virtualized environments. It abstracts devices to the guest domains through simple interfaces. This abstraction ensures a fair amount of security and is highly efficient in terms of performance as data transfer occurs using shared memory. Whenever an interrupt occurs, Xen's lightweight event delivery system sends asynchronous notifications to a domain. These notifications are sent through asynchronous buffer descriptor rings. A typical I/O descriptor ring is a circular queue of descriptors allocated by a domain but accessible from within Xen. Descriptors do not directly contain I/O data; instead, I/O data buffers are allocated out of band by the guest OS and indirectly referenced by I/O descriptors. Access to each ring is based around two pairs of producer-consumer pointers: domains place requests on a ring, advancing a request producer pointer, and Xen removes these requests for handling, advancing an associated request consumer pointer. Responses are placed back on the ring similarly, save with Xen as the producer and the guest OS as the consumer. There is no requirement that requests be processed in order: the guest OS associates a unique identifier with each request which is reproduced in the associated response. This allows Xen to unambiguously reorder I/O operations due to scheduling or priority considerations. This structure is sufficiently generic to support a number of different device paradigms.

Xen has introduced the concept of Virtual Firewall Router (VFR) where each domain can afford to have more than one virtual network interfaces (VIF). These interfaces abstract the network hardware card which is accoutered with two asynchronous I/O rings of buffer descriptors for bi-directional I/O. For each direction of I/O that occurs, there exist specific rules which state the appropriate actions to be taken whenever a pattern of operation is noticed. These rules are inserted, removed and altered through the control interface to the hypervisor. They are implemented in common POSIX compatible iptables [x] directives. While transmitting a packet, a guest domain enqueues a buffer descriptor into the I/O ring meant for transmission. Hypervisor copies the buffer descriptor and in turn the packet header from this ring and is responsible for deciding where to route the packet depending on any rules specified. Packet payload is then sent using DMA. Depending on the rules specified at the virtual firewall router, packets are either discarded or forwarded to the domains. While receiving, guest domain allocates an unused page frame for each packet that appears. If the packet is destined correctly to arrive at a guest domain, then packets are copied to the buffer in the receiving I/O ring.

The mechanism for disk I/O is somewhat similar to network I/O in Xen. The control interface to the Xen hypervisor or Domain 0 has full privileged access to the physical disks. Guest domains can only access the disk through virtual block devices (VBD) which are totally managed from

domain 0. Each virtual block device consists of a list of extents. These extents are logical units that have their own access control information and are accessed via the I/O ring mechanism. Disk schedulers implemented within the guest domains, reorder requests that involve operations at contiguous location in the disk before queuing them. This saves significant amount of response time. However, as the hypervisor has a thorough knowledge of the disk geometry, reordering requests within it is a preferable option. The hypervisor maintains a translation table for each VBDs whose contents are modified by a management console only accessible by domain 0. While receiving a disk request, the hypervisor generates the actual sector address with the help of the guest domain identifier and the offset furnished by the requesting domain. The data transfer occurs through DMA. Requests are batched and executed in a round robin manner

Observations

There are a lot of positives for Xen and it presents a very strong case for itself in the server consolidation space.

- ☒ Xen is a robust para virtualization solution with less than 50K lines of code.
- ☒ Performance figures of Xen are very attractive and Xen domains offer near native performance for both CPU and IO intensive applications.
- ☒ Scalable to a large number of partitions/domains with very little loss in performance
- ☒ Advanced resource control mechanisms in Xen enable better SLAs
- ☒ Supports live migration of Xen domain with zero down time
- ☒ Robust architecture guarantees strong isolation between domains
- ☒ Xen is Open Source licensed under GNU GPL version 2
- ☒ Xen has made its way into the latest Fedora distribution of Linux and is likely to be integrated into other distributions in very near future
- ☒ Intel, IBM, Red Hat and other big names sponsor and contribute to Xen development and maintenance.
- ☒ Xen has ports available for AMD 64 Bit processor, EMT64 Bit Processor from Intel and also Itanium apart from the IA32 architecture.
- ☒ Xen supports Intel's VT and AMD Pacifica packaged with their latest 32 and 64 Bit processors. With VT & Pacifica unmodified Windows and other proprietary OS can be loaded in a Xen domain
- ☒ Support up to 32Way SMP servers
- ☒ Support beyond 4GB Memory access for 32Bit processor

However, there are a few downsides to Xen,

- ☒ Kernel need to be ported to Xen architecture when run on processors without virtualization support (VT & Pacifica). This may not be possible with proprietary Operating Systems like Microsoft Windows.
- ☒ Unlike commercial offerings like VMWare ESX Server [4] and SWSOft Virtuozzo [5], the support structure for Xen is not sound yet. XenSource is an effort to address this issue. XenSource [6] has recently announced release of XenOptimizer [6], a virtual infrastructure management tool for data centers and support subscription. Also when Xen gets into main stream Linux distributions like Red Hat, support will be available for Xen through Red Hat and other Linux vendors.
- ☒ Linux, with a licensing and distribution model similar to Xen is likely to go well together, compared to Xen for MS Windows or other operating system despite the fact that processor may support virtualizing proprietary operating systems in future. This however is not a serious shortcoming, as it is common knowledge that Linux is making a big stride into the enterprises and large enterprises are set to migrate their computing platforms from MS Windows to Linux.
- ☒ With newer virtualization support from the processors, products like VMWare ESX Server are likely to have near native performance. This may challenge Xen's position in the market. Xen's synergy with Intel and Linux is likely to hold it in good stead.

Conclusion

Xen with its low performance overhead and high scalability can help enterprises consolidate their services onto fewer physical servers, improving their overall resource utilization and also cutting down on their maintenance budgets. Xen's strong isolation, fine resource control can provide firm SLA guarantees. Live migration feature in Xen can help IT administrators to plan maintenance of their servers without disruption of services to their customers and other stakeholders. This can also improve the overall reliability of the services hosted in the data centers. Xen's support for high-end SMP servers would help consolidate several small to medium services onto a single physical server.

References

1. Paul Barham , Boris Dragovic , Keir Fraser , Steven Hand , Tim Harris , Alex Ho , Rolf Neugebauer , Ian Pratt , Andrew Warfield, "Xen and the art of virtualization", Proceedings of the nineteenth ACM symposium on Operating systems principles, October 1922, 2003, Bolton Landing, NY, USA
2. Intel Virtualization Technology, <http://www.intel.com/technology/computing/vptech>
3. AMD Pacifica, Source: <http://enterprise.amd.com/Enterprise/serverVirtualization.aspx>
4. Mendel Rosenblum, Tal Garfinkel. "Virtual Machine Monitors: Current Technology and Future Trends," Computer, vol. 38, no. 5, pp. 3947, May 2005.
5. Virtuozzo: "A Complete Server Virtualization and Automation Solution {Product White Paper & Data Sheet}". Source: <http://www.swsoft.com/>
6. XenSource and XenOptimizer, Source: <http://www.xensource.com/>

© 2006 Infosys Technologies Limited.

ALL RIGHTS RESERVED

Copyright in whole and in part of this document "Xen and Server Consolidation" belongs to Infosys Technologies Limited. This work may not be used, sold, transferred, adapted, abridged, copied or reproduced in whole or in part in any manner or form or in any media without the prior written consent of Infosys Technologies Limited.

Pg 10

InfiniBand Device Virtualization in Xen

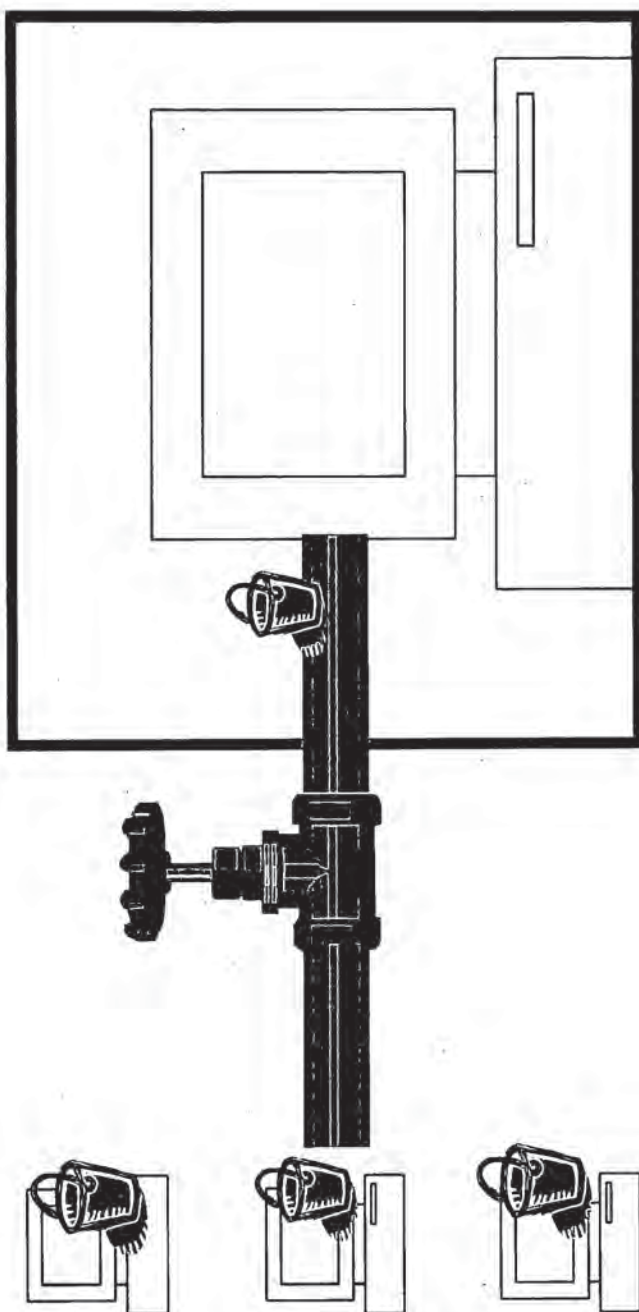
Dror Goldenberg (gdror@mellanox.co.il)

January 19, 2006

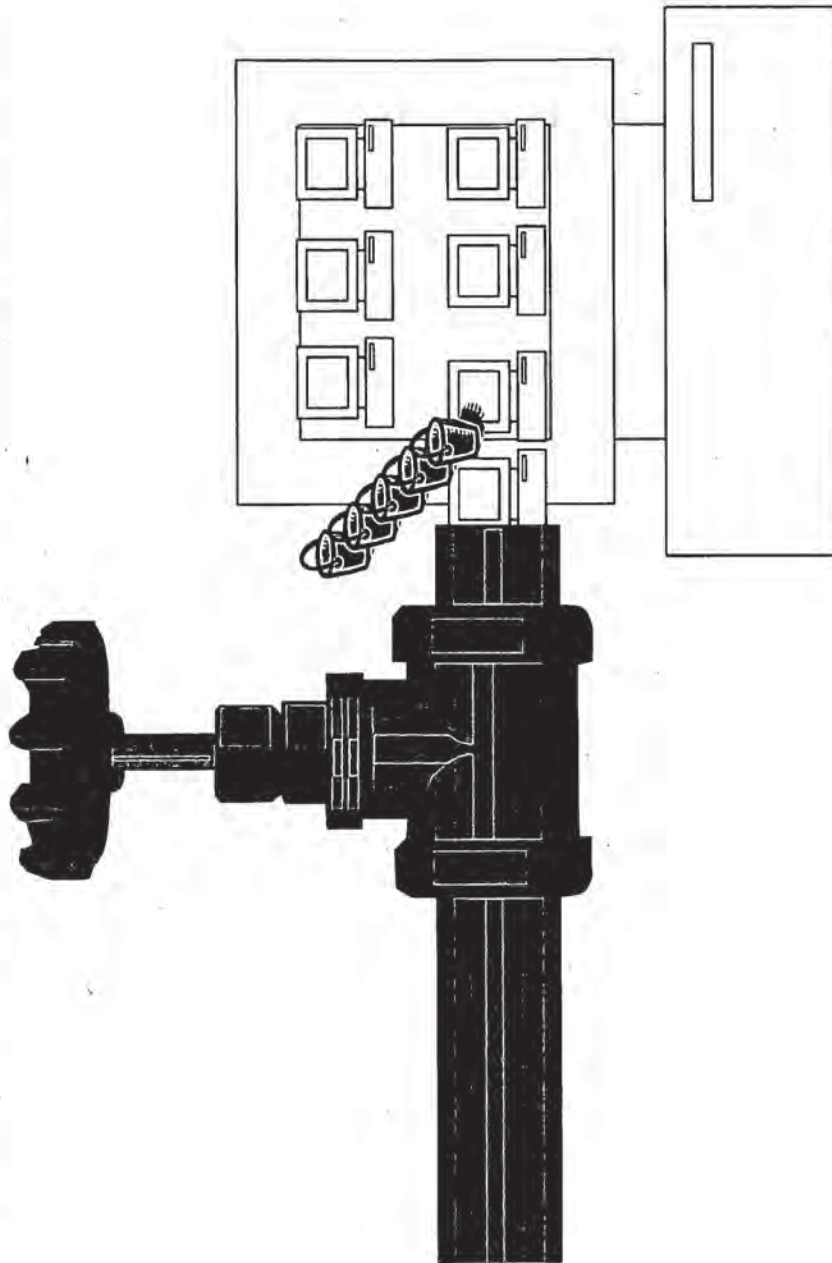




The IO Bottleneck



The IO Bottleneck (Virtualization)



InfiniBand Key Benefits

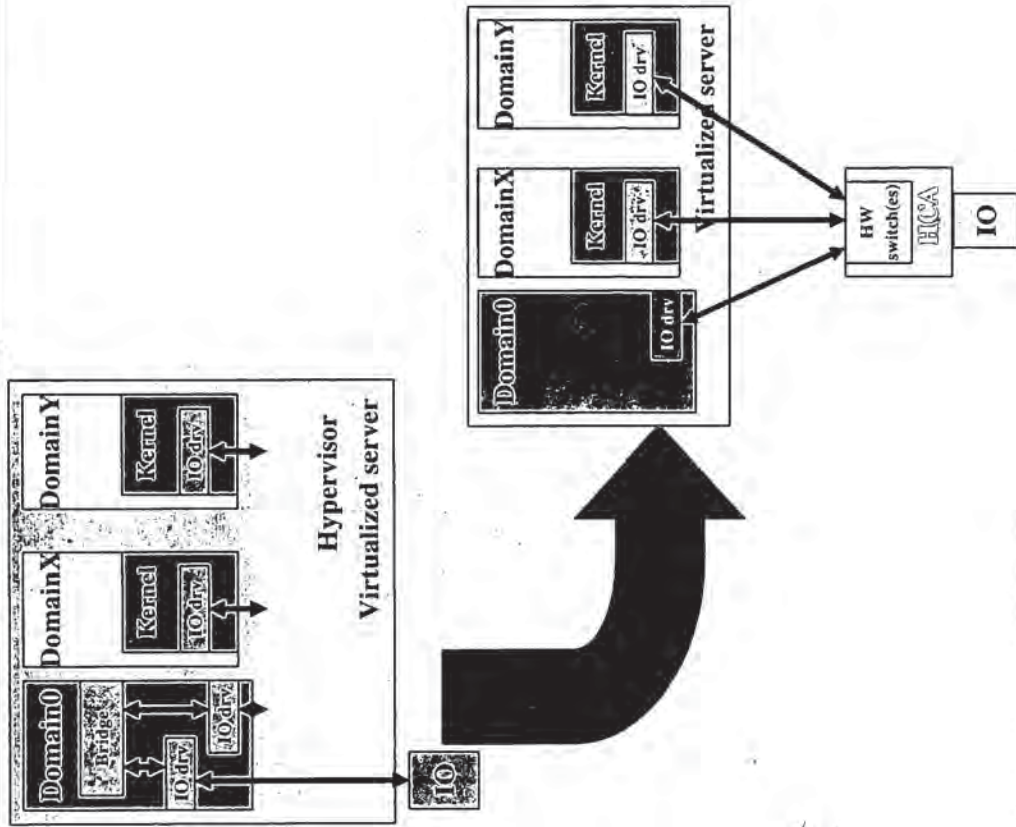


- Take InfiniBand Characteristics to the Virtualized Environment
 - Unmatchable I/O
 - Low Latency (<2.5us application to application)
 - High Bandwidth (>2600MB/s bidirectional bandwidth)
 - CPU Offloads
 - Transport offload
 - RDMA
 - Kernel (and Hypervisor) bypass
 - Fabric Consolidation
 - IPC, Network, Storage, Backup, Management
- Mature SW Stack
 - OpenIB stack is part of the Linux kernel

InfiniBand Key Benefits (Cont'd)



- Channel Based I/O
 - Cross-channel isolation
 - Cross-channel protection
 - Native device sharing
- Hypervisor and Virtualization Stack Offloads
 - Saves data copies
 - Reduces context switching
 - Virtual switching
- Existing HW Fully Supports Virtualization
 - The most cost-effective path for single-node virtual servers
 - SW-transparent scale-out



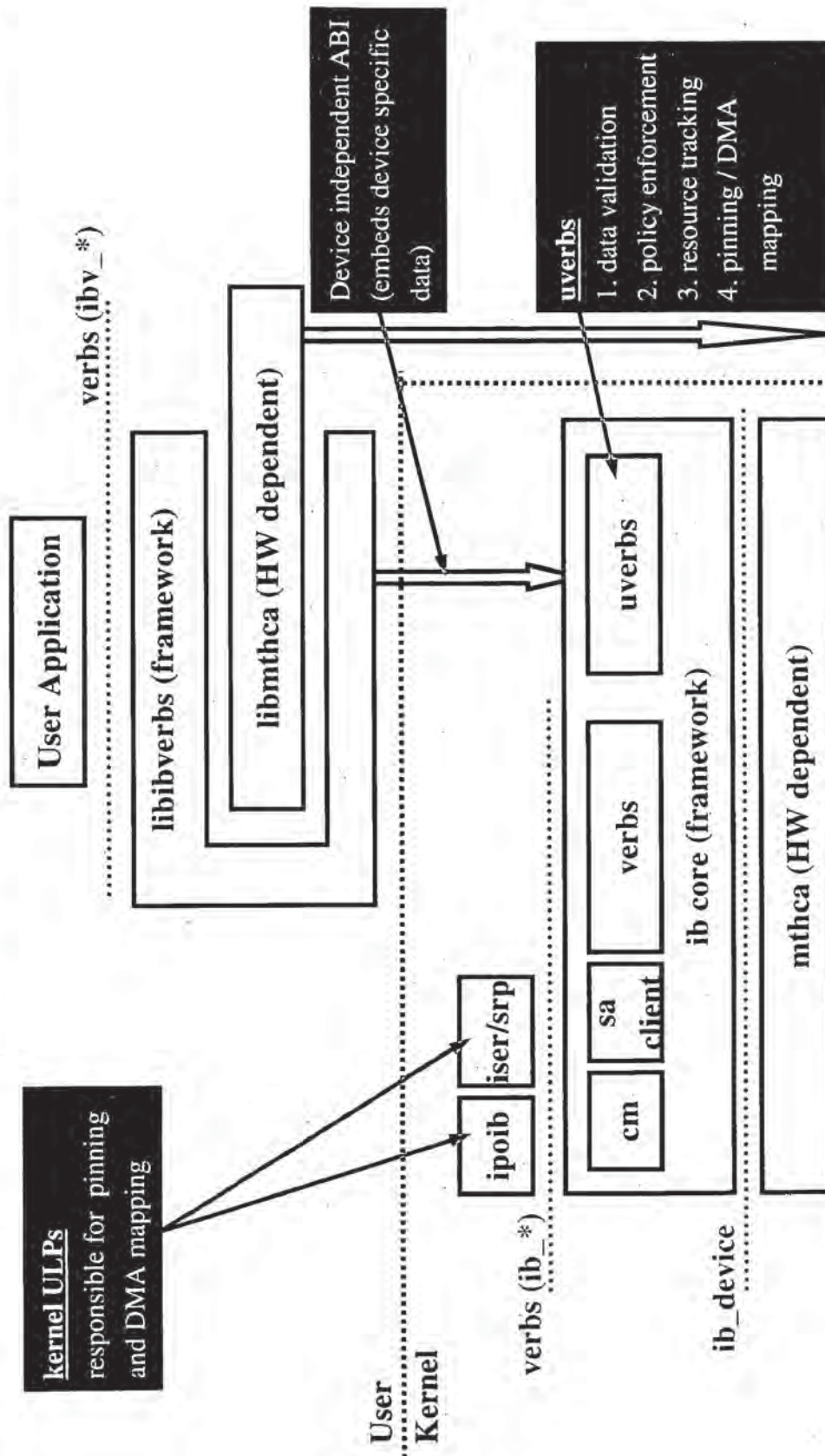
Goals



- **Leverage high-performance IO in VMM**
 - VM-transparent Hypervisor offload
- **Direct access to IO from guest VM**
 - Full Hypervisor bypass
 - Match native InfiniBand performance on virtual machine
- **Enable InfiniBand unaware guests**
- **Multiple solutions: performance vs. HW awareness tradeoff**
 - Partial InfiniBand performance gain – DomU is HW independent
 - Full InfiniBand performance gain – DomU is HW dependent
 - Different customers need different solutions

- **Solution I: Enable IB in Xen Environment**
 - Networking – IPoIB through standard net front/back
 - Block Storage – SRP/iSER through standard block front/back
- **Solution II (1st Part): Native IB Networking and Storage in DomU**
 - IB Front/Back infrastructure
 - Networking – IPoIB in Dom U over IB access layer
 - Storage – SRP/iSER in Dom U over IB access layer
- **Solution II (2nd Part): Full Native IB Support in DomU**
 - SDP
 - File storage (NFS/RDMA)
 - Userland access layer
 - MPI
- **Solutions can operate concurrently**

OpenIB Linux Stack Architecture

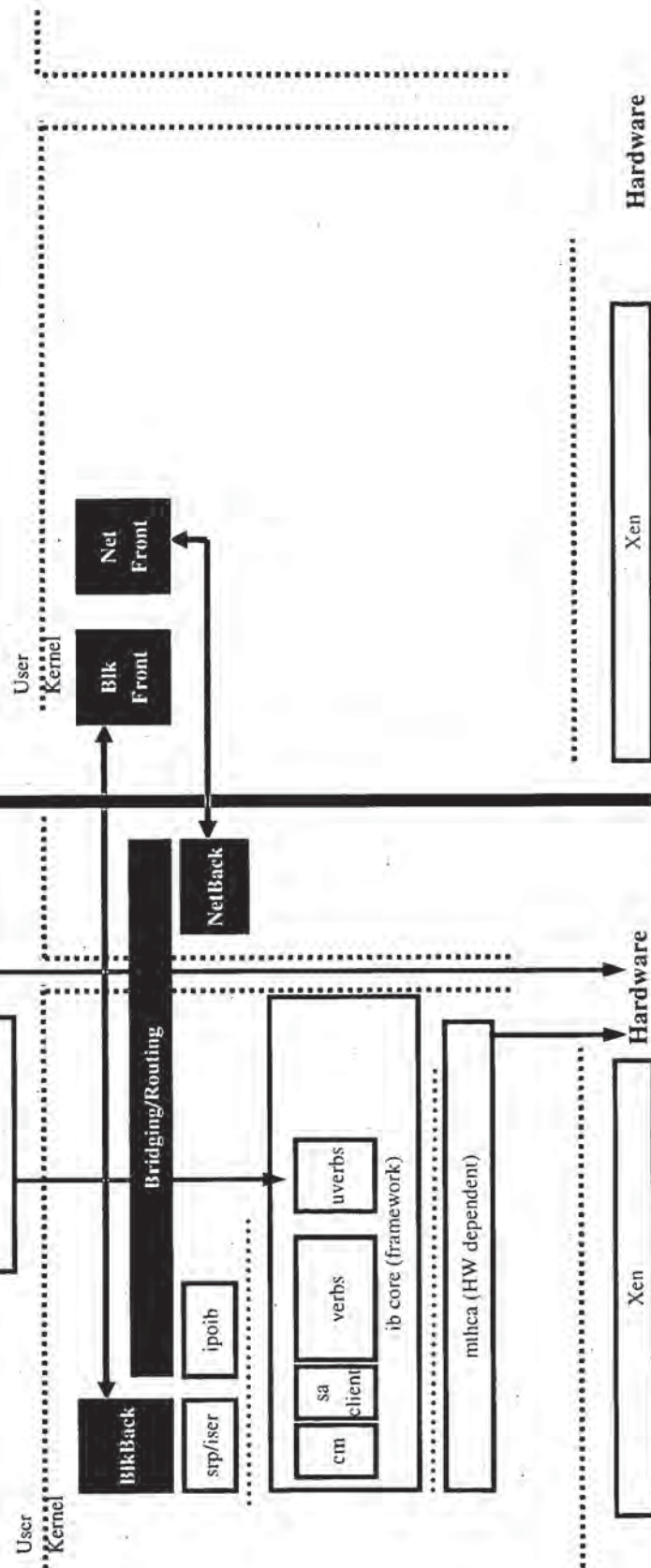




Solution I: Infrastructure

DomU

Dom0

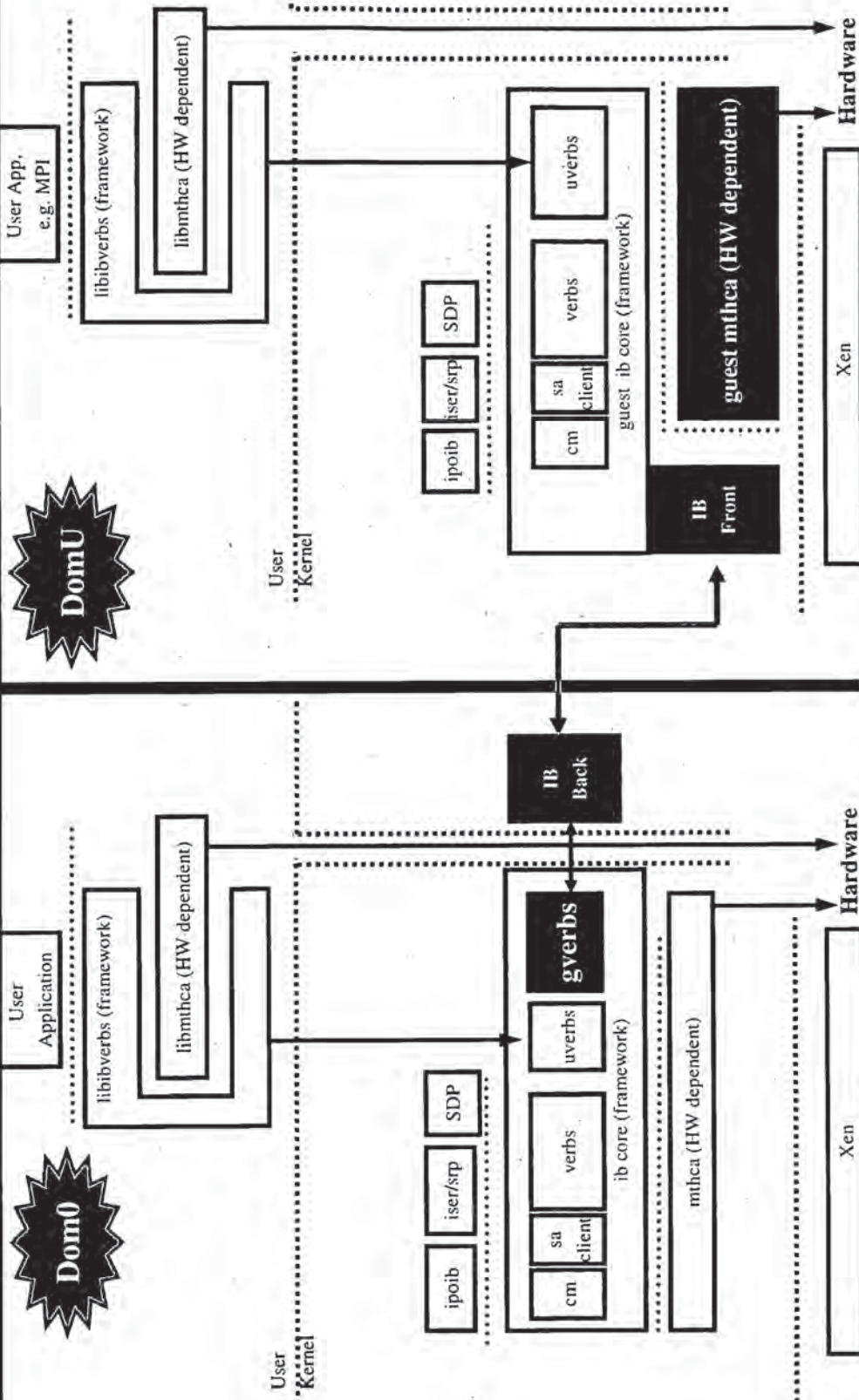


Solution I: Summary



- **Benefits**
 - Xen machine can connect to InfiniBand infrastructure
 - Utilize InfiniBand high performance on network/storage
 - Can be further improved using Solution II
 - **Hardware independent Dom U**
 - Can enable running legacy OS guest that has no IB support
- **Challenges**
 - **IPoIB MAC address is 20B; alternatives to packet demultiplexing (Dom 0):**
 - use multiple QPs in Dom0 and assign one QP per DomU
 - use 20B MAC address all the way netfront/back/bridge
 - use IP based demultiplexing at Dom 0
 - **Multicast requires explicit registration to groups**
 - **No issues expected with storage**

Solution II: InfiniBand Frontend / Backend



Solution II (1st Part): Summary



- **Benefits**
 - Bring native IB to the guest kernel
 - Improve network/storage performance
 - Expect close to native performance
 - Reduction in context switch, packet copy, etc.
 - Enable InfiniBand HW independent Dom U
- **Challenges**
 - **IPoIB uses physical unregistered addresses**
 - Patch kernel to use pre-registered networking data buffers
 - Register entire DomU physical address
 - Need to exclude PTE/Balloon pages
 - Separate send and receive pinning method
 - Send – use read-only registered memory – can also be used for PTEs
 - Receive – use special allocator that registers buffers for DMA write
 - **SRP uses physical addresses**
 - Use HCA paging on demand
 - Batch registration
 - Use paging on demand
 - **Interrupt forwarding to DomU**

Solution II (2nd Part): Summary



- **Benefits**
 - Bring native IB to the guest
 - Improve network performance - SDP
 - Enable middleware running in DomU – e.g. MPI

High Performance VMM-Bypass I/O in Virtual Machines

JIUXING LIU, WEI HUANG, BULENT ABALI, AND DHABALESWAR K. PANDA

Technical Report
OSU-CISRC-2/06-TR22

High Performance VMM-Bypass I/O in Virtual Machines *

Jiuxing Liu

Wei Huang[†]

Bulent Abali

Dhabaleswar K. Panda[†]

IBM T. J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532
{jl, abali}@us.ibm.com

[†] Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{huanwei, panda}@cse.ohio-state.edu

Abstract

Currently, I/O device virtualization models in virtual machine (VM) environments require involvement of a virtual machine monitor (VMM) and/or a privileged VM for each I/O operation, which may turn out to be a performance bottleneck for systems with high I/O demands, especially those equipped with modern high speed interconnects such as InfiniBand.

In this paper, we propose a new device virtualization model called VMM-bypass I/O, which extends the idea of OS-bypass originated from user-level communication. Essentially, VMM-bypass allows time-critical I/O operations to be carried out directly in guest VMs without involvement of the VMM and/or a privileged VM. By exploiting intelligence found in modern high speed network interfaces, VMM-bypass can significantly improve I/O and communication performance for VMs without sacrificing safety or isolation.

To demonstrate the idea of VMM-bypass, we have developed a prototype called Xen-IB, which offers InfiniBand virtualization support in the Xen 3.0 VM environment. Xen-IB runs with current InfiniBand hardware and does not require modifications to existing user-level applications or kernel-level drivers that uses InfiniBand. Our performance measurements show that the Xen-IB is able to achieve nearly the same raw performance as the original InfiniBand running in a non-virtualized environment.

1 Introduction

Virtual machine (VM) technologies were first introduced in the 1960s [14]. Recently they are experiencing a resurgence and becoming more and more attractive to both the industry and the research communities [35]. A key com-

ponent in a VM environment is the virtual machine monitor (VMM) (also called hypervisor), which is implemented directly on top of hardware and provides virtualized hardware interfaces to VMs. With the help of VMMs, VM technologies allow running many different virtual machines in a single physical box, with each virtual machine possibly running a different operating system. VMs can also provide secure and portable environments to meet the demanding requirements of computing resources in modern computing systems [8].

In VM environments, device I/O access in guest operating systems is handled in different ways by various VM technologies. For instance, in VMware Workstation, device I/O relies on switching back to the host operating system and user-level emulation [37]. In VMware ESX Server, guest VM I/O operations trap into the VMM, which makes direct access to I/O devices [44]. In Xen [10], a high performance virtual machine monitor originally developed at the University of Cambridge, device I/O follows a split-driver model. Only an isolated device domain (IDD) has access to the hardware using native device drivers. All other virtual machines (guest VMs or domains) need to pass the I/O requests to the IDD to access those devices. This control transfer between domains needs involvement of the VMM.

In recent years, network interconnects that provide very low latency (less than 10 μ s) and very high bandwidth (multiple Gbps) are emerging. Some examples of these high speed interconnects include Virtual Interface Architecture (VIA) [11], InfiniBand [19], Quadrics [34], and Myrinet [25]. Due to their excellent performance, these interconnects have become stronger players in areas such as high performance computing. To achieve high performance, these interconnects usually have intelligent network interface cards (NICs) which can be used to offload a large part of the host communication protocol processing. The intelligence in the NICs also supports user-level communication, which enables safe direct I/O access from user-level processes (OS-bypass I/O) and contributes to reduced la-

*This research is supported in part by Department of Energy's Grant #DE-FC02-01ER25506 and National Science Foundation grants #CNS-0403342 and #CCR-0509452.

tency and CPU overhead.

VM technologies can greatly benefit computing systems built from the aforementioned high speed interconnects by not only simplifying cluster management for these systems, but also offering much cleaner solutions to tasks such as check-pointing and fail-over. Recently, as these high speed interconnects become more and more commoditized with their cost going down, they are also used for remote I/O access in high-end enterprise systems, which increasingly run in virtualized environments. Therefore, it is very important to provide VM support to high-end systems equipped with these high speed interconnects. However, performance and scalability requirements of these systems pose some challenges. In all the VM I/O access approaches mentioned previously, VMMs have to be involved to make sure that I/O accesses are safe and do not compromise integrity of the system. Therefore, current device I/O access in virtual machines requires context switches between the VMM and guest VMs. Thus, I/O access can suffer from longer latency and higher CPU overhead compared to native I/O access in non-virtualized environments. In some cases, the VMM may also become a performance bottleneck which limits I/O performance in guest VMs. In some of the aforementioned approaches (VM Workstation and Xen), a host operating system or another virtual machine is also involved in the I/O access path. Although these approaches can greatly simplify VMM design by moving device drivers out of the VMM, they may lead to even higher I/O access overhead when requiring context switches between the host operating system and the guest VM or two different VMs, which are fairly expensive.

In this paper, we present a *VMM-bypass* approach for I/O access in VM environments. Our approach takes advantages of features found in modern high speed intelligent network interfaces to allow time-critical operations to be carried out directly in guest VMs while still maintaining system integrity and isolation. With this method, we can remove the bottleneck of going through the VMM or a separate VM for many I/O operations and significantly improve communication and I/O performance. The key idea of our VMM-bypass approach is based on the OS-bypass design of these high speed network interfaces, which allows user processes to access I/O devices directly in a safe way without going through operating systems. OS-bypass was originally proposed by research communities [43, 42, 29, 5, 33] and later adopted by many commercial interconnects such as InfiniBand. Our idea can be regarded as an extension of OS-bypass designs in the context of VM environments.

To demonstrate the idea of VMM-bypass, we have designed and implemented a prototype called *Xen-IB* to provide virtualization support for InfiniBand in the Xen VM environment. Basically, our implementation presents to each guest VM a paravirtualized InfiniBand device. Our

design requires no modification to existing hardware. Also, through a technique called *high-level virtualization*, we allow current user-level applications and kernel-level modules that utilize InfiniBand to run without modification. Our performance results, which includes benchmarks at the basic InfiniBand level as well as evaluation of upper-layer InfiniBand protocols such as IP over InfiniBand (IPoIB) [1] and MPI [36], demonstrate that performance of our VMM-bypass approach comes close to that in a native, non-virtualized environment. Although our current implementation is for InfiniBand and Xen, the basic VMM-bypass idea and many of our implementation techniques can be readily applied to other high-speed interconnects and other VMMs.

In summary, the main contributions of our work are:

- We proposed the VMM-bypass approach for I/O accesses in VM environments for modern high speed interconnects. Using this approach, many I/O operations can be performed directly without involvement of a VMM or another VM. Thus, I/O performance can be greatly improved.
- Based on the idea of VMM-bypass, we implemented a prototype, *Xen-IB*, to virtualize InfiniBand devices in Xen guest VMs. Our prototype supports running existing InfiniBand applications and kernel modules in guest VMs without modification.
- We carried out extensive performance evaluation of our prototype. Our results show that performance of our virtualized InfiniBand device is very close to native InfiniBand devices running in a non-virtualized environment.

The rest of the paper is organized as follows: In Section 2, we present background information, including the Xen VM environment and the InfiniBand architecture. In Section 3, we present the basic idea of VMM-bypass I/O. In Section 4, we discuss detailed design and implementation of our *Xen-IB* prototype. In Section 5, we discuss several related issues and limitations of our current implementation and how they can be addressed in future. Performance evaluation results are given in Section 6. We discuss related work in Section 7 and conclude the paper in Section 8.

2 Background

In this section, we provide background information for our work. In Section 2.1, we describe how I/O device accesses are handled in several popular VM environments. In Section 2.3, we describe the OS-bypass feature in modern high speed network interfaces. Since our prototype is based on Xen and InfiniBand, we introduce them in Sections 2.2 and 2.4, respectively.

2.1 I/O Device Access in Virtual Machines

In a VM environment, the VMM plays the central role of virtualizing hardware resources such as CPUs, memory, and I/O devices. To maximize performance, the VMM needs to let guest VMs access these resources directly whenever possible. Take CPU virtualization as an example, a guest VM can execute all non-privileged instructions natively in hardware without intervention of the VMM. However, privileged instructions executed in guest VMs will generate a trap into the VMM. The VMM will then take necessary steps to make sure that the execution can continue without compromising system integrity. Since many CPU intensive workloads seldom use privileged instructions (This is especially true for applications in HPC area.), they can achieve excellent performance even when executed in a VM.

I/O device access in VMs, however, is a completely different story. Since I/O devices are usually shared among all VMs in a physical machine, the VMM has to make sure that accesses to them are legal. Currently, this requires VMM intervention on every I/O access from guest VMs. For example, in VMware ESX Server [44], all physical I/O accesses are carried out within the VMM, which includes device drivers for popular server hardware. System integrity is achieved with every I/O access going through the VMM. Furthermore, the VMM can serve as an arbitrator/multiplexer/demultiplexer to implement useful features such as QoS control among VMs. However, VMM intervention also leads to longer I/O latency and higher CPU overhead due to the context switches between guest VMs and the VMM. Since the VMM serves as a central control point for all I/O accesses, it may also become a performance bottleneck for I/O intensive workloads.

Having device I/O access in the VMM also complicates the design of the VMM itself. It significantly limits the range of supported physical devices because new device drivers have to be developed to work within the VMM. To address this problem, VMware workstation [37] and Xen [12] carry out I/O operations in a host operating system or a special privileged VM called isolated device domain (IDD), which can run popular operating systems such as Windows and Linux that have a large number of existing device drivers. Although this approach can greatly simplify the VMM design and increase the range of supported hardware, it does not directly address performance issues with the approach used in VMware ESX Server. In fact, I/O accesses now may result in expensive operations called a world switch (a switch between the host OS and the VMs) or a domain switch (a switch between different VMs), which can lead to even worse I/O performance.

2.2 Overview of the Xen Virtual Machine Monitor

Xen is a popular high performance VMM. It uses para-virtualization [45], in which host operating systems need to

be explicitly ported to the Xen architecture. This architecture is similar to native hardware such as the x86 architecture, with only slight modifications to support efficient virtualization. Since Xen does not require changes to the application binary interface (ABI), existing user applications can run without any modification.

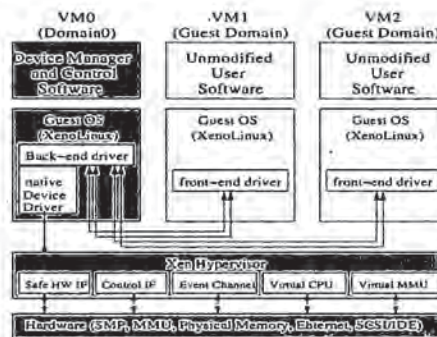


Figure 1. The structure of the Xen hypervisor, hosting three xenoLinux operating systems (courtesy [32])

Figure 1 illustrates the structure of a physical machine running Xen. The Xen hypervisor is at the lowest level and has direct access to the hardware. The hypervisor, instead of the guest operating systems, is running in the most privileged processor-level. Xen provides basic control interfaces needed to perform complex policy decisions in Xen architecture. Above the hypervisor are the Xen domains (VMs). There can be many domains running simultaneously. Guest OSes are prevented from directly executing privileged processor instructions. A special domain called *domain0*, which is created at boot time, is allowed to access the control interface provided by the hypervisor. The guest OS in *domain0* hosts the application-level management software and perform the tasks to create, terminate or migrate other domains through the control interface.

There is no guarantee that a domain will get a continuous stretch of physical memory to run a guest OS. So Xen makes a distinction between *machine memory* and *pseudo-physical memory*. Machine memory refers to the physical memory installed in the machine, while pseudo-physical memory is a per-domain abstraction, allowing a guest OS to treat its memory as a contiguous range of physical pages. Xen maintains the mapping between the machine and the pseudo-physical memory. Only a certain parts of the operating system needs to understand the difference between these two abstractions. Guest OSes allocate and manage their own hardware page tables, with minimal involvement of the Xen hypervisor to ensure safety and isolation.

In Xen, domains communicate with each other through

shared pages and *event channels*. Event channels provide an asynchronous notification mechanism between domains. Each domain has a set of end-points (or ports) which may be bounded to an event source [40]. When a pair of end-points in two domains are bound together, a “send” operation on one side will cause an event to be received by the destination domain, which may in turn cause an interrupt. Event channels are only intended for sending notifications between domains. So if a domain wants to send data to another, the typical scheme is for a source domain to grant access to local memory pages to the destination domain. Then, these shared pages are used to transfer data.

Virtual machines in Xen usually do not have direct access to hardware. Since most existing device drivers assume they have complete control of the device, there cannot be multiple instantiations of such drivers in different domains for a single device. To ensure manageability and safe access, device virtualization in Xen follows a split device driver model [13]. Each device driver is expected to run in an *isolated device domain (IDD)*, which also hosts a *backend* driver, running as a daemon and serving the access requests from guest domains. Each guest OS uses a *frontend* driver to communicate with the backend. The split driver organization provides security: misbehaving code in a guest domain will not result in failure of other guest domains. Since the split device driver model requires the development of frontend and backend drivers for each individual device, only a limited number of devices are currently supported in guest domains.

2.3 OS-bypass I/O

Traditionally, device I/O accesses are carried out inside the OS kernel on behalf of application processes. However, this approach poses several problems such as overhead caused by context switches between user processes and OS kernels and extra data copies which degrade I/O performance [4]. It can also result in *QoS crosstalk* [17] due to lacking of proper accounting for costs of I/O accesses carried out by the kernel on behalf of applications.

To address these problems, a concept called user-level communication was introduced by the research community. One of the notable features of user-level communication is *OS-bypass*, with which I/O (communication) operations can be achieved directly by user processes without involvement of OS kernels. OS-bypass was later adopted by commercial products, many of which have become popular in areas such as high performance computing where low latency is vital to applications. It should be noted that OS-bypass does not mean all I/O operations bypass the OS kernel. Usually, devices allow OS-bypass for frequent and time-critical operations while other operations, such as setup and management operations, can go through OS kernels and are handled by a privileged module, as illustrated in Figure 2.

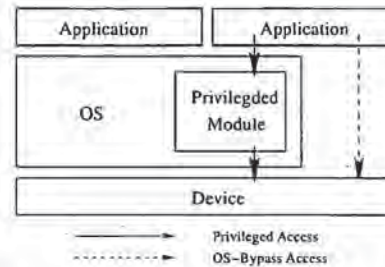


Figure 2. OS-Bypass Communication and I/O

The key challenge to implement OS-bypass I/O is to enable safe accesses to a device shared by many different applications. To achieve this, OS-bypass capable devices usually require more intelligence in the hardware than traditional I/O devices. Typically, an OS-bypass capable device is able to present virtual access points to different user applications. Hardware data structures for virtual access points can be encapsulated into different I/O pages. With the help of an OS kernel, the I/O pages can be mapped into the virtual address spaces of different user processes. Thus, different processes can access their own virtual access points safely, thanks to the protection provided by the virtual memory mechanism. Although the idea of user-level communication and OS-bypass was developed for traditional, non-virtualized systems, the intelligence and self-virtualizing characteristic of OS-bypass devices lend themselves nicely to a virtualized environment, as we will see later.

2.4 InfiniBand Architecture

InfiniBand [19] is a high speed interconnect offering high performance as well as features such as OS-bypass. InfiniBand host channel adapters (HCAs) are the equivalent of network interface cards (NICs) in traditional networks. A queue-based model is used in the interface presented to the consumers. A *Queue Pair (QP)* in the InfiniBand Architecture consists of a send queue and a receive queue. The send queue holds instructions to transmit data and the receive queue holds instructions that describe where received data is to be placed. Communication operations are described in *Work Queue Requests (WQR)*, or descriptors, and submitted to the queue pairs. The completion of the communication is reported through *Completion Queues (CQ)* by *Completion Queue Entries (CQEs)*. An application can subscribe for notifications from HCA and register a callback handler with CQ. Completion queue can also be accessed through polling to reduce latency.

Initiating data transfer (posting work requests) and completion of work requests notification (polling for completion) are time-critical tasks which uses OS-bypass. In the Mellanox [22] approach, which represents a typical imple-

mentation of InfiniBand specification, these operations are done by ringing a doorbell. Doorbells are rung by writing to the registers that form the *User Access Region (UAR)*. Each UAR is a 4k I/O page mapped into a process's virtual address space. It allows access to HCA resources from privileged as well as unprivileged mode. Posting a work request includes putting the descriptors (WQR) to QP buffer and writing the doorbell to the UAR, which is completed without the involvement of the operating system. CQ buffers, where the CQEs are located, can also be directly accessed from the process virtual address space. These OS-bypass features make it possible for InfiniBand to provide very low communication latency.

InfiniBand also provides a comprehensive management scheme. Management communication is achieved by sending management datagrams (MADs) to well-known QPs (QP0 and QP1).

InfiniBand requires all buffers involved in communication be registered before they can be used in data transfers. In Mellanox HCAs, the purpose of registration is two-fold. First, an HCA needs to keep an entry in the Translation and Protection Table (TPT) so that it can perform virtual-to-physical translation and protection checks during data transfer. Second, the memory buffer needs to be pinned in memory so that HCA can DMA directly into the target buffer. Upon the success of registration, a local key and a remote key are returned. They will be used later for local and remote (RDMA) accesses. QP and CQ buffers described above are just normal buffers that are directly allocated from process virtual memory space and registered with HCA.

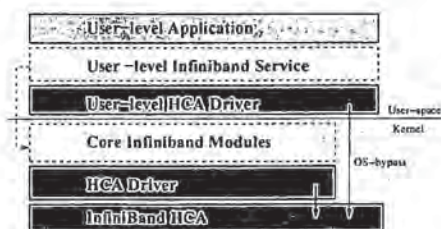


Figure 3. Architectural overview of OpenIB Gen2 stack

There are two popular stacks for InfiniBand drivers. VAPI [23] is the Mellanox implementation and OpenIB Gen2 [28] recently have come out as a new generation of IB stack provided by the OpenIB community. In this paper, our prototype implement is based on OpenIB Gen2, whose architecture is illustrated in Figure 3.

3 VMM-Bypass I/O

VMM-bypass I/O can be viewed as an extension to the idea of OS-bypass I/O in the context of VM environments. In this section, we describe the basic design of VMM-bypass I/O. Two key ideas in our design are *paravirtualization* and *high-level virtualization*.

In some VM environments, I/O devices are virtualized at the hardware level [37]. Each I/O instruction to access a device is virtualized by the VMM. With this approach, existing device drivers can be used in the guest VMs without modification. However, the method significantly increases the complexity of virtualizing devices. For example, one popular InfiniBand card (MT23108 from Mellanox [24]) presents itself as a PCI-X device to the system. After configuration, this device can be accessed by the OS using memory mapped I/O. Virtualizing this device at the hardware level would require us to not only understand all the hardware commands issued through memory mapped I/O, but also implement a virtual PCI-X bus in the guest VM. Another problem with this approach is performance. Since existing physical devices are typically not designed to run in a virtualized environment, the interfaces presented at the hardware level may exhibit significant performance degradation when they are virtualized.

Our VMM-bypass I/O virtualization design is based on the idea of paravirtualization, similar to [10] and [46]. We do not preserve hardware interfaces of existing devices. To virtualize a device in a guest VM, we implement a device driver called *guest module* in the OS of the guest VM. The guest module is responsible for handling all the privileged accesses to the device. In order to achieve VMM-bypass device access, the guest module also needs to set things up properly so that I/O operations can be carried out directly in the guest VM. This means that the guest module must be able to create virtual access points on behalf of the guest OS and map them into the addresses of user processes. Since the guest module does not have direct access to the device hardware, we need to introduce another software component called *backend module*, which provides device hardware access for different guest modules. If devices are accessed inside the VMM, the backend module can be implemented as part of the VMM. It is possible to let the backend module talk to the device directly. However, we can greatly simplify its design by reusing the original privilege module of the OS-bypass device driver. In addition to serving as a proxy for device hardware access, the backend module also coordinates accesses among different VMs so that system integrity can be maintained. The VMM-bypass I/O design is illustrated in Figure 4.

If device accesses are provided by another VM (device driver VM), the backend module can be implemented within the device driver VM. The communication between

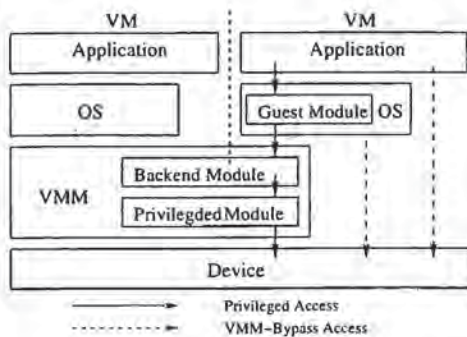


Figure 4. VM-Bypass I/O (I/O Handled by VMM Directly)

guest modules and the backend module can be achieved through the inter-VM communication mechanism provided by the VM environment. This approach is shown in Figure 5.

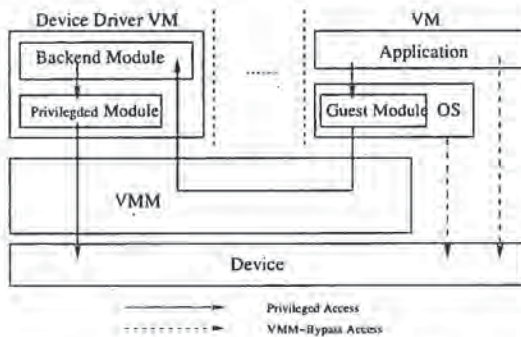


Figure 5. VM-Bypass I/O (I/O Handled by Another VM)

Paravirtualization can lead to compatibility problems because a paravirtualized device does not conform to any existing hardware interfaces. However, in our design, these problems can be addressed by maintaining existing interfaces which are at a higher level than the hardware interface (a technique we dubbed *high-level virtualization*). Modern interconnects such as InfiniBand and VIA have their own standardized access interfaces. For example, InfiniBand specification defines a *VERBS* interface for a host to talk to an InfiniBand device. The VERBS interface is usually implemented in the form of an API set through a combination of software and hardware. Our high-level virtualization approach maintains the same VERBS interface within a guest VM. Therefore, existing kernel drivers and applications that

use InfiniBand will be able to run without modification. Although in theory a driver or an application can bypass the VERBS interface and talk to InfiniBand devices directly, this seldom happens because it leads to poor portability due to the fact that different InfiniBand devices may have different hardware interfaces.

4 Prototype Design and Implementation

In this section, we present the design and implementation of Xen-IB, our InfiniBand virtualization driver for Xen. We describe details of the design, which follows the Xen split device driver model, and how we enable accessing the HCA from guest domains directly for time-critical tasks.

4.1 Overview

Like many other device drivers, InfiniBand drivers cannot have multiple instantiations for a single HCA. Thus a split driver model approach is required to share a single HCA among multiple Xen domains.

Figure 6 illustrates a basic design of our Xen-IB driver. The backend runs as a kernel daemon on top of the native InfiniBand driver in the isolated device domain (IDD), which is domain0 in our current implementation. It waits for incoming requests from the frontend drivers in the guest domains. The frontend driver, which corresponds to the guest module mentioned in Section 3, replaces the kernel HCA driver in OpenIB Gen2 stack. Once the frontend is loaded, it establishes two event channels with the backend daemon. The first channel, together with shared memory pages, forms a device channel [13] which is used to process the requests initiated from the guest domain. The second channel is used for sending InfiniBand CQ and QP events to the guest domain and will be discussed in detail later.

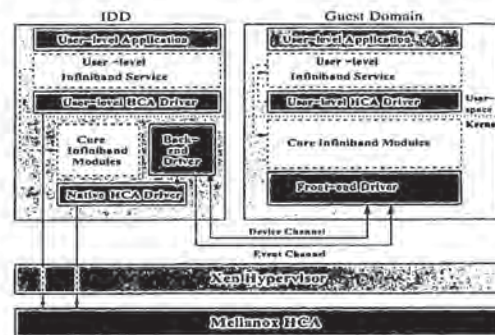


Figure 6. The Xen-IB driver structure with the split driver model

The Xen-IB frontend driver provides the same set of interfaces as normal Gen2 stack for kernel modules. It is a

relatively thin layer whose tasks include packing the command together with necessary parameters and sending it to the backend through the device channel. The backend driver reconstructs the commands, performs the operations with the native kernel HCA driver on behalf of the guest domain, and returns the result to the frontend driver. The backend manages InfiniBand resources on behalf of guest domains. Only handles to those resources are passed back to the frontend driver for later references.

The split device driver model in Xen poses difficulties for user-level direct HCA access in Xen guest domains. To enable VMM-bypass, we need to let guest domains have direct access to certain HCA resources such as the UARs and the QP/CQ buffers.

4.2 InfiniBand Privileged Accesses

In the following, we discuss in general how we support all privileged InfiniBand operations, including initialization, InfiniBand resource management, memory registration and event handling.

Initialization and resource management: Before applications can send messages using InfiniBand, it must finish several preparation steps including opening HCA, creating CQ, creating QP, and modifying QP status, etc. Those operations are usually not in the time critical path of the applications and can be implemented in a straightforward way. Basically, the guest domains forward these commands to the device domain (IDD) and wait for the acknowledgments after the operations are completed in IDD. All the resources are managed in IDD and the frontends refer to these resources by handles. Validation checks must be conducted in IDD to ensure that all references are legal.

Memory Registration: The InfiniBand specification requires all the memory regions involved in data transfers be registered with HCA. With Xen's para-virtualization approach, all domains see the same DMA address as the real machine address (assuming that no IOMMU is used). So there is no extra need for address translation. The information needed by memory registration is a list of DMA addresses that describes the physical locations of the buffers, access flags and the virtual address that the application will use when accessing the buffers. Again, the registration happens at the device domain. The frontend driver sends above information to the backend driver and get back the local and remote keys. Note that since the Translation and Protection Table (TPT) on HCA is indexed by keys, multiple guest domains are allowed to register with the same virtual address.

For security reasons, the backend driver can verify if the frontend driver offers valid DMA addresses belonging to the specific domain that it is running in. This check will make sure that all later communication activities of guest domains are within the valid address spaces.

Event Handling: InfiniBand supports several kinds of CQ and QP events. The most commonly used is the completion event, which is introduced in section 2.4. Event handlers are associated with CQs or QPs when they are created. An application can subscribe for event notification by writing the appropriate command to the UAR page. When those subscribed events happen, the HCA driver will first get notified from HCA and then dispatch the event to different CQs or QPs according to the event type. Then the application that owns the CQ/QP will get a callback on their event handlers.

For Xen-IB, events are generated for the device domain, where all QPs and CQs are actually created. But the device domain cannot directly give a callback to the event handlers in the guest domains. To address this issue, we create a dedicated event channel between a frontend and the backend driver. The backend driver associates a special event handler to each CQ/QP created due to requests from guest domains. Each time the HCA generates an event to these CQs/QPs, this special event handler gets executed and forwards the information such as the event type and the CQ/QP identifier to the guest domain through the event channel. The frontend driver binds an event dispatcher as a callback handler to one end of the event channel after the channel is created. The event handlers given by the applications are associated to the CQs or QPs after they are successfully created. Frontend driver also maintains a translation table between the CQ/QP identifiers and the actual CQ/QPs. Once the event dispatcher gets an event notification from the backend driver, it checks the identifier and gives the corresponding CQ/QP a callback on the associated handler.

4.3 VMM-Bypass Accesses

In InfiniBand, QP accesses (posting descriptors) include writing WQEs to the QP buffer and ringing the doorbell to notify the HCA. Then the HCA can use DMA to transfer the WQEs to internal HCA memory and perform the send/receive or RDMA operations. Once a work request is completed, HCA will put a completion entry (CQE) in the CQ buffer. In InfiniBand, QP access functions are used for initiating communication. To detect completion of communication, CQ polling can be used. QP access and CQ polling functions are typically found in the critical path of communication. Therefore, it is very important to optimize their performance by using VMM-bypass. The basic architecture of the VMM-bypass design is shown in Figure 7.

Supporting VMM-bypass for QP access and CQ polling poses two requirements on our design of Xen-IB driver: first, the UAR page must be accessible from a guest domain; second, both QP and CQ buffers should be directly visible in the guest domain.

When a frontend driver is loaded, the backend driver allocates a UAR page and returns its page frame number

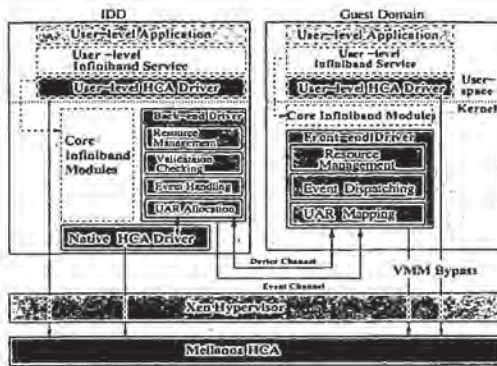


Figure 7. VMM-Bypass design of Xen-IB driver

(machine address) to the frontend. The frontend driver then remaps this page to its own address space so that it can directly access the UAR in the guest domain to serve requests from the kernel applications¹. In the same way, when a user application starts, the frontend driver also applies for a UAR page from the backend and remaps the page to the application's virtual memory address space, which can be later accessed directly from the user space. Since all UARs are managed in a centralized manner in the IDD, there will be no conflicts between UARs in different guest domains.

To make QP and CQ buffers accessible to guest domains, creating CQs/QPs has to go through two stages. In the first stage, QP or CQ buffers are allocated in the guest domains and registered through the IDD. During the second stage, the frontend sends the CQ/QP creation commands to the IDD along with the keys returned from the registration stage to complete the creation process. Address translations are indexed by keys, so in later operations the HCA can directly read WQRs from and write the CQEs back to the buffers located in the guest domains.

Since we also allocate UARs to user space applications in guest domains, the user level InfiniBand library now keeps its OS-bypass feature. The VMM-bypass IB-Xen workflow is illustrated in Figure 8.

4.4 Virtualizing InfiniBand Management Operations

In an InfiniBand network, management and administrative tasks are achieved through the use of Management Datagrams (MADs). MADs are sent and received just like normal InfiniBand communication, except that they must use two well-known queue-pairs: QP0 and QP1. Since there is only one set of such queue pairs in every HCA, their access must be virtualized for accessing from many different

¹ We have applied a small patch to Xen to enable access to I/O pages in guest domains.

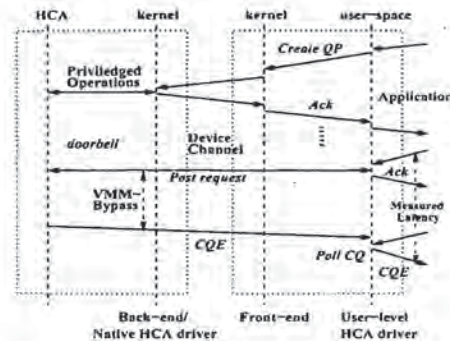


Figure 8. Working flow of the VMM-bypass Xen-IB driver

VMs, which means we must treat them differently than normal queue-pairs. However, since queue-pair accesses are done directly in guest VMs in our VMM-bypass approach, it would be very difficult to track each queue-pair access and take different actions based on whether it is a management queue-pair or a normal one.

To address this difficulty, we use the idea of high-level virtualization. This is based on the fact that although MAD is the basic mechanism for InfiniBand management, applications and kernel drivers seldom use it directly. Instead, different management tasks are achieved through more user-friendly and standard API sets which are implemented on top of MADs. For example, the kernel IPoIB protocol makes use of the subnet administration (SA) services, which are offered through a high-level, standardized SA API. Therefore, instead of tracking each queue-pair access, we virtualize management functions at the API level by providing our own implementation for guest VMs. Most functions can be implemented in a similar manner as privileged InfiniBand operations, which typically includes sending a request to the backend driver, executing the request (backend), and getting a reply. Since management functions are rarely in time-critical paths; the implementation will not bring any significant performance degradation. However, it does require us to implement every function provided by all the different management interfaces. Fortunately, there are only a couple of such interfaces and the implementation effort is not significant.

5 Discussions

In this section, we discuss issues related to our prototype implementation such as how safe device access is ensured, how performance isolation between different VMs can be achieved, and challenges in implementing VM check-pointing and migration with VMM-bypass. We also point out several limitations of our current prototype and

how we can address them in future.

5.1 Safe Device Access

To ensure that accesses to virtual InfiniBand devices by different VMs will not compromise system integrity, we need to make sure that both privileged accesses and VMM-bypass accesses are safe. Since all privileged accesses need to go through the backend module, access checks are implemented there to guarantee safety. VMM-bypass operations are achieved through accessing the memory-mapped UAR pages which contain virtual access points. Setting-up these mappings is privileged and can be checked. InfiniBand allows using both virtual and physical addresses for sending and receiving messages or carrying out RDMA operations, as long as a valid memory key is presented. Since the key is obtained through InfiniBand memory registration, which is also a privileged operation, we implement necessary safety checks in the backend module to ensure that a VM can only carry out valid memory registration operations. It should be noted that once a memory buffer is registered, its physical memory pages cannot be reclaimed by the VMM. Therefore, we should limit the total size of buffers that can be registered by a single VM. This limit check can also be implemented in the backend module.

Memory registration is an expensive operation in InfiniBand. In our virtual InfiniBand implementation, memory registration cost is even higher due to inter-domain communication. This may lead to performance degradation in cases where buffers cannot be registered in advance. Techniques such as *pin-down cache* can be applied when buffers are reused frequently, but it is not always effective. To address this issue, some existing InfiniBand kernel drivers create and use a *DMA key* through which all physical pages can be accessed. Currently, our prototype supports DMA keys. However, this leaves a security hole because all physical memory pages (including those belonging to other VMs) can be accessed. In future, we plan to address this problem by let DMA keys only authorize access to physical pages in the current VM. However, this also means that we need to update the keys whenever the VMM changes the physical pages allocated to a VM.

5.2 Performance Isolation

Although our current prototype does not yet implement performance isolation or QoS among different VMs, this issue can be addressed by taking advantage of QoS mechanisms which are present in the current hardware. For example, Mellanox InfiniBand HCAs support a QoS scheme in which a weighted round-robin algorithm is used to schedule different queue-pairs. In this scheme, QoS policy parameters are assigned when queue-pairs are created and initialized. After that, the HCA hardware is responsible for taking necessary steps to ensure QoS policies. Since queue-pair

creations are privileged, we can create desired QoS policies in the backend when queue-pairs are created. These QoS policies will later be enforced by device hardware. We plan to explore more along this direction in future.

5.3 VM Check-pointing and Migration

VMM-bypass I/O poses new challenges for implementing VM check-pointing and migration. This is due to two reasons. First, the VMM does not have complete knowledge of VMs with respect to device accesses. This is in contrast to traditional device virtualization in which the VMM is involved in every I/O operation and it can easily suspend and buffer these operations when check-pointing or migration starts. The second problem is that VMM-bypass I/O exploits intelligent devices which can store a large part of the VM system states. For example, an InfiniBand HCA has onboard memory which stores information such as registered buffers, queue-pair data structures, and so on. Some of the state information on an HCA can only be changed as side effects of VERBS functions calls. It does not allow changing it in an arbitrary way. This makes it difficult for check-pointing and migrations because when a VM is restored from a previous checkpoint or migrated to another node, the corresponding state information on the HCA needs to be restored also.

There are two directions to address the above problems. The first one is to involve VMs in the process of check-pointing and migration. For example, the VMs can bring themselves to some determined states which simplify check-pointing and migration. Another way is to introduce some hardware/firmware changes. We are currently working on both directions.

6 Performance Evaluation

In this section, we first evaluate the performance of our Xen-IB prototype using a set of micro-benchmarks at the InfiniBand layer. Then, we present performance results for the IPoIB protocol. We also provide performance numbers for MPI at both micro-benchmark and application levels.

6.1 Experimental Setup

Our experimental testbed is an InfiniBand cluster. Each system in the cluster is equipped with dual Intel Xeon 3.0GHz CPUs, 2 GB memory and a Mellanox MT23108 PCI-X InfiniBand HCA. The PCI-X bus on these systems were 64 bit and ran at 133 MHz. The systems are connected with an InfiniScale InfiniBand switch. The operating systems we used are RedHat AS4 with kernel 2.6.12. We used Xen 3.0 for all our experiments. All Xen guest domains run with a single virtual CPU and 512 MB memory.

6.2 InfiniBand Latency and Bandwidth

In this subsection, we compared user-level latency and bandwidth performance between Xen-IB and native Infini-

Band. Xen-IB results were obtained from two guest domains running in two different physical machines. Polling was used for detecting completion of communication.

The latency tests were carried out in a ping-pong fashion. The tests were repeated for many times and the average half round-trip time was reported as one-way latency. Figures 9 and 10 show the results for InfiniBand RDMA write and send/receive operations, respectively. We can see that there is very little performance difference between Xen-IB and native InfiniBand. This is because in the tests, all InfiniBand communication operations were carried out by directly accessing the HCA from guest domains with VMM-bypass. The lowest latency achieved by both was around 4.2 μ s for RDMA write and 6.6 μ s for send/receive.

In the bandwidth tests, a sender sent a number of messages to a receiver and then waited for an acknowledgment. The bandwidth was obtained by dividing the number of bytes transferred from the sender to the receiver by the elapsed time of the test. From Figures 11 and 12, we again see virtually no difference between Xen-IB and native InfiniBand. Both of them were able to achieve bandwidths up to 880 MByte/s, which was limited by the bandwidth of the PCI-X bus.

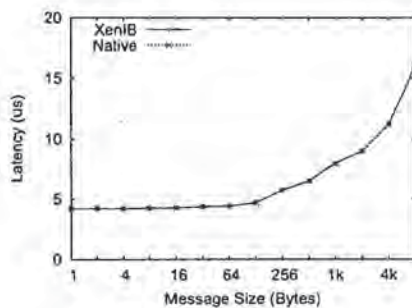


Figure 9. InfiniBand RDMA Write Latency

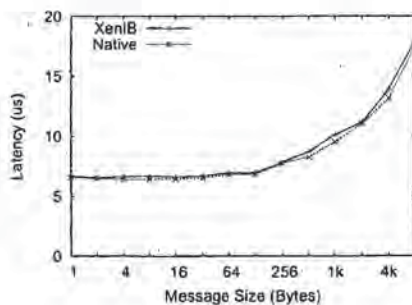


Figure 10. InfiniBand Send/Receive Latency

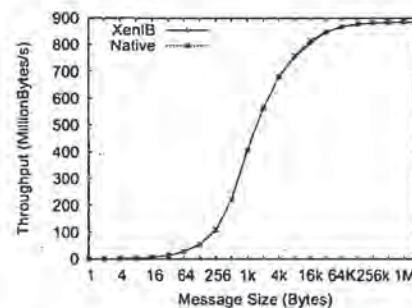


Figure 11. InfiniBand RDMA Write Bandwidth

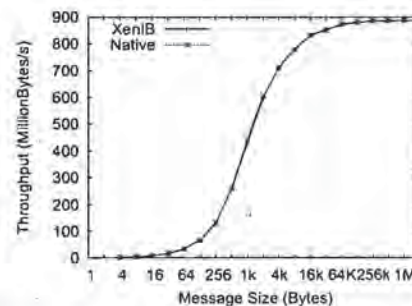


Figure 12. InfiniBand Send/Receive Bandwidth

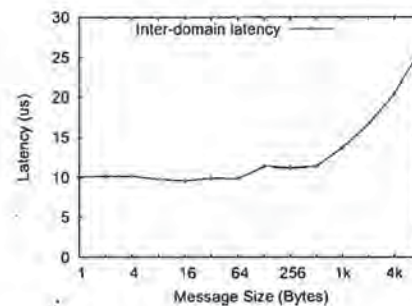


Figure 13. Inter-domain Communication One Way Latency

6.3 Event/Interrupt Handling Overhead

The latency numbers we showed in the previous subsection were based on polling schemes. In this section, we characterize the overhead of event/interrupt handling in Xen-IB by showing send/receive latency results with blocking InfiniBand user-level VERBS functions.

Compared with native InfiniBand event/interrupt processing, Xen-IB introduces extra overhead because it requires forwarding an event from domain0 to a guest domain, which involves Xen inter-domain communication. In Figure 13, we show performance of Xen inter-domain communication. We can see that the overhead increases with the amount of data transferred. However, even with very small messages, there is an overhead of about 10 μ s.

Fig. 14 shows the send/receive one-way latency using blocking VERBS. The test is almost the same as the send/receive latency test using polling. The difference is that a process will block and wait for a completion event instead of busy polling on the completion queue. From the figure, we see that Xen-IB has higher latency due to overhead caused by inter-domain communication. For each message, Xen-IB needs to use inter-domain communication twice, one for send completion and one for receive completion. For large messages, we observe that the difference between Xen-IB and native InfiniBand is around 18–20 μ s, which is roughly twice the inter-domain communication latency. However, for small messages, the difference is much less. For example, native InfiniBand latency is only 3 μ s better for 1 byte messages. This difference gradually increases with message sizes until it reaches around 20 μ s. Our profiling reveals that this is due to “event batching”. For small messages, the inter-domain latency is much higher than InfiniBand latency. Thus, when a send completion event is delivered to a guest domain, a reply may have already come back from the other side. Therefore, the guest domain can process two completions with a single inter-domain communication operation, which results in reduced latency. For small messages, event batching happens very often. As message size increases, it becomes less and less frequent and the difference between Xen-IB and native IB increases.

6.4 Memory Registration

Memory registration is generally a costly operation in InfiniBand. Figure 15 shows the registration time of Xen-IB and native InfiniBand. The benchmark registers and unregisters a trunk of user buffers multiple times and measures the average time for each registration.

As we can see from the graph, Xen-IB adds consistently around 25%-35% overhead to the registration cost. The overhead increases with the number of pages involved in registration. This is because Xen-IB needs to use inter-domain communication to send a message which contains

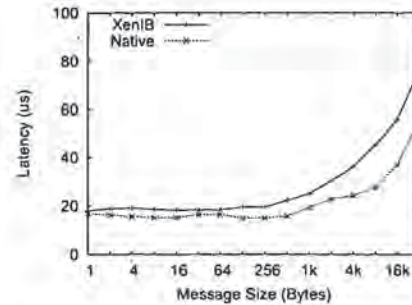


Figure 14. Send/Receive Latency Using Blocking VERBS Functions

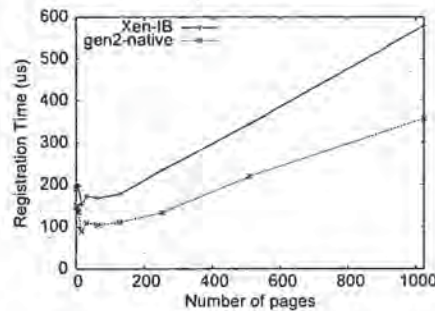


Figure 15. Memory Registration Time

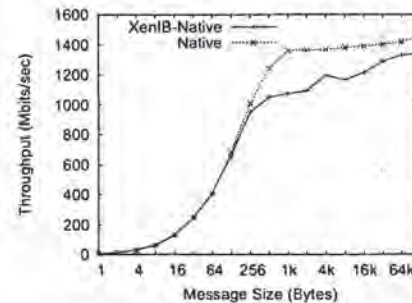


Figure 16. IPoIB Netperf Throughput

machine addresses of all the pages. The more pages we register, the bigger the size of message we need to send to the device domain through the inter-domain device channel. This observation indicates that if the registration is a time critical operation of an application, we need to use techniques such as an efficient implementation of registration cache [39] to reduce costs.

6.5 IPoIB Performance

IPoIB allows one to run TCP/IP protocol suites over InfiniBand. In this subsection, we compared IPoIB performance between Xen-IB and native InfiniBand using Netperf [2]. In all the Netperf tests, we used a guest domain with IB-Xen as the server. The client was running with native InfiniBand.

Figure 16 shows the results for TCP throughput tests in Netperf. Due to increased cost of interrupt/event processing, Xen-IB does not perform as well as native InfiniBand. However, it is able to reach more than 90% the throughput achieved by native InfiniBand for large messages.

We notice that IPoIB achieved much less bandwidth compared with raw InfiniBand. This is because of two reasons. First, IPoIB uses InfiniBand unreliable datagram service, which has significantly lower bandwidth than the more frequently used reliable connection service due to the current implementation of Mellanox HCAs. Second, in IPoIB, due to the limit of MTU, large messages are divided into small packets, which can cause a large number of interrupts and degrade performance.

Figure 17 shows the results of Netperf transaction test. Again, Xen-IB performs worse than native InfiniBand, especially for small messages where interrupt/event cost plays a dominant role for performance. Xen-IB performs more comparable to native InfiniBand for large messages.

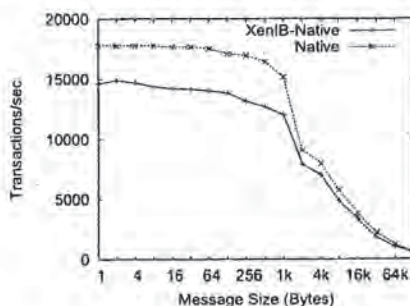


Figure 17. Netperf Transaction Test

6.6 MPI Performance

MPI is a communication protocol used in high performance computing. For tests in this subsection, we have

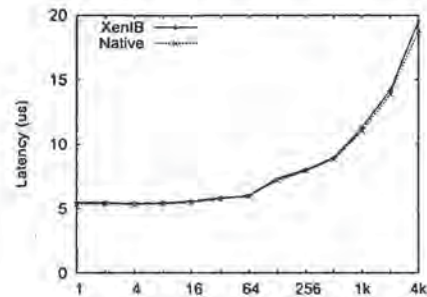


Figure 18. MPI Latency

used MVAPICH [27, 21], which is a popular MPI implementation over InfiniBand.

Figures 18 and 19 compare Xen-IB and native InfiniBand in terms of MPI one-way latency and bandwidth. The tests were run between two physical machines in the cluster. Since MVAPICH uses polling for all underlying InfiniBand communication, Xen-IB was able to achieve the same performance as native InfiniBand by using VMM-bypass. The smallest latency achieved by MPI with Xen-IB was 5.4 μ s. The peak bandwidth was 870 MBytes/s.

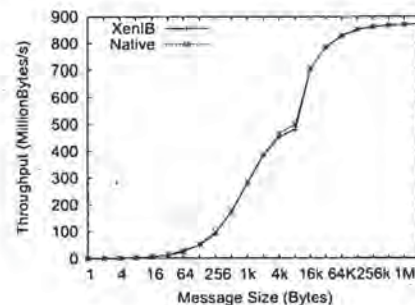


Figure 19. MPI Bandwidth

Figure 20 shows performance of IS, FT, SP and BT applications from the NAS Parallel Benchmarks suite [26] (class A), which is frequently used by researchers in the area of high performance computing. We show normalized execution time based on native InfiniBand. In these tests, two physical nodes were used with two guest domains per node for Xen-IB. For native InfiniBand, two MPI processes were launched for each node. We can see that Xen-IB performs comparably with native InfiniBand, even for communication intensive applications such as IS. IB-Xen performs about 4% worse for FT and around 2-3% better for SP and BT. We believe the difference is due to the fact that MVAPICH uses shared memory communication for processes in a single node. Although MVAPICH with Xen-IB currently

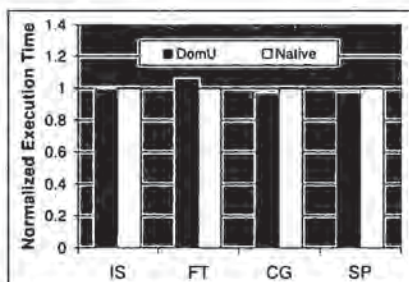


Figure 20. MPI NAS Benchmarks

does not have this feature, it can be easily added by taking advantage of the page sharing mechanism provided by Xen.

7 Related Work

In Section 2.1, we have discussed current I/O device virtualization approaches such as those in VMware Workstation [37], VMware ESX Server [44], and Xen [12]. All of them require the involvement of the VMM or a privileged VM to handle every I/O operation. In our VMM-bypass approach, many time-critical I/O operations can be executed directly by guest VMs. Since this method makes use of intelligence in modern high speed network interfaces, it is limited to a relatively small range of devices which are used mostly in high-end systems. The traditional approaches can be applied to a much wider ranges of devices.

OS-bypass is a feature found in user-level communication protocols such as active messages [43], U-Net [42], FM [29], VMMC [5], and Arsenic [33]. Later, it was adopted by the industry [11, 19] and found its way into commercial products [25, 34]. Our work extends the idea of OS-bypass to VM environments. With VMM-bypass, I/O and communication operations can be initiated directly by user space applications, bypassing the guest OS, the VMM, and the device driver VM. VMM-bypass also allows an OS in a guest VM to carry out many I/O operations directly, although virtualizing interrupts still needs the involvement of the VMM.

The idea of direct device access from a VM has been proposed earlier. For example, [6] describes a method to implement direct I/O access from a VM for IBM mainframes. However, it requires an I/O device to be dedicated to a specific VM. The VMM-bypass approach not only enables direct device access, but allows for safe device sharing among many different VMs. Recently, the industry has started working on standardization of I/O virtualization by extending the PCI Express standard [30] to allow a physical device to present itself as multiple virtual devices to the

system [31]. This approach can potentially allow a VM to directly interact with a virtual device. However, it requires building new hardware support into PCI devices while our VMM-bypass approach is based on existing hardware. At about the same time when we were working on our virtualization support for InfiniBand in Xen, others in the InfiniBand community proposed similar ideas [41, 38]. However, details regarding their implementations are currently not available.

Our InfiniBand virtualization support for Xen uses a paravirtualization approach. As a technique to improve VM performance by introducing small changes in guest OSes, paravirtualization has been used in many VM environments [7, 16, 46, 10]. Essentially, paravirtualization presents a different abstraction to the guest OSes than native hardware, which lends itself to easier and faster virtualization. The same idea can be applied to the virtualization of both CPU and I/O devices. Paravirtualization usually trades compatibility for enhanced performance. However, our InfiniBand virtualization support achieves both high performance and good compatibility by maintaining the same interface as native InfiniBand drivers at a higher level than hardware. As a result, our implementation is able to support existing kernel drivers and user applications. Virtualization at higher levels than native hardware is used in a number of other systems. For example, novel operating systems such as Mach [15], K42 [3], and L4 [18] use OS level API or ABI emulation to support traditional OSes such as Unix and Linux. Several popular VM projects also used this approach [20, 9].

8 Conclusions and Future Work

In this paper, we presented the idea of VMM-bypass, which allows time-critical I/O commands to be processed directly in guest VMs without involvement of a VMM or a privileged VM. VMM-bypass can significantly improve I/O performance in VMs by eliminating context switching overhead between a VM and the VMM or two different VMs caused by current I/O virtualization approaches. To demonstrate the idea of VMM-bypass, we described the design and implementation of Xen-IB, an VMM-bypass capable InfiniBand driver for the Xen VM environment. Xen-IB runs with current InfiniBand hardware and does not require modification to applications or kernel drivers which use InfiniBand. Our performance evaluations showed that Xen-IB can provide performance close to native hardware under most circumstances, with expected degradation on event/interrupt handling and memory registration.

Currently, we are working on providing check-pointing and migration support for our Xen-IB prototype. We are also investigating how to provide performance isolation by implementing QoS support in Xen-IB. In future, we plan

to study the possibility to introduce VMs into high performance computing area. We will explore how to take advantages of Xen to provide better support of check-pointing, QoS and cluster management with minimum loss of computing power.

Acknowledgments

We would like to thank Charles Schulz, Dan Poff, Mohammad Banikazemi, and Scott Guthridge of IBM Research for valuable discussions and their support.

References

- [1] IP over InfiniBand Working Group. <http://www.ietf.org/himl.charters/ipoib-charter.html>.
- [2] Netperf. <http://www.netperf.org>.
- [3] Jonathan Appavoo, Marc Auslander, Maria Burtico, Dilma Da Silva, Orran Krieger, Mark Mergen, Michal Ostrowski, Bryan Rosenberg, Robert W. Wisniewski, and Jimi Xenidis. K42: an Open-Source Linux-Compatible Scalable Operating System Kernel. *IBM Systems Journal*, 44(2):427–440, 2005.
- [4] R. A. F. Bhoedjang, T. Ruhl, and H. E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, pages 53–60, November 1998.
- [5] M. Blumrich, C. Dubnicki, E. W. Felten, K. Li, and M. R. Mesarina. Virtual-Memory-Mapped Network Interfaces. In *IEEE Micro*, pages 21–28, Feb. 1995.
- [6] T. L. Borden, J. P. Hennessy, and J. W. Rymarczyk. Multiple Operating Systems on One Processor Complex. *IBM System Journal*, 28(1):104–123, 1989.
- [7] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.
- [8] P. M. Chen and B. D. Noble. When virtual is better than real. *Hot Topics in Operating Systems*, pages 133–138, 2001.
- [9] Dan Aloni. Cooperative Linux. <http://www.colinux.org>.
- [10] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 164–177, October 2003.
- [11] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A.M. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, pages 66–76, March/April 1998.
- [12] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield and M. Williamson. Safe hardware access with the xen virtual machine monitor. In *Proceedings of OASIS ASPLOS Workshop*, 2004.
- [13] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Reconstructing I/O. Technical Report UCAM-CL-TR-596, University of Cambridge, UK, August 2004.
- [14] R. P. Goldberg. Survey of Virtual Machine Research. *Computer*, pages 34–45, June 1974.
- [15] David B. Golub, Randall W. Dean, Alessandro Forin, and Richard F. Rashid. UNIX as an application program. In *USENIX Summer*, pages 87–95, 1990.
- [16] Kingshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 18(3):229–262, 2000.
- [17] Steven M. Hand. Self-paging in the nemesis operating system. In *Operating Systems Design and Implementation, USENIX*, pages 73–86, 1999.
- [18] H. Hartig, M. Hohmuth, J. Liedtke, and S. Schonberg. The performance of micro-kernel-based systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 66–77, December 1997.
- [19] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.2.
- [20] Jeff Dike. User Mode Linux. <http://user-mode-linux.sourceforge.net>.
- [21] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *Proceedings of 17th Annual ACM International Conference on Supercomputing (ICS '03)*, June 2003.
- [22] Mellanox Technologies. <http://www.mellanox.com>.
- [23] Mellanox Technologies. Mellanox IB-Verbs API (VAPI), Rev. 1.00.
- [24] Mellanox Technologies. Mellanox InfiniBand InfiniHost MT23108 Adapters. <http://www.mellanox.com>, July 2002.
- [25] Myricom, Inc. Myrinet. <http://www.myri.com>.
- [26] NASA. NAS Parallel Benchmarks. <http://www.nas.nasa.gov/Software/NPB/>.
- [27] Network-Based Computing Laboratory. MVAPICH: MPI for InfiniBand on VAPI Layer. <http://nowlab.cis.ohio-state.edu/projects/mpi-iba/index.html>.
- [28] Open InfiniBand Alliance. <http://www.openib.org>.
- [29] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM). In *Proceedings of the Supercomputing*, 1995.
- [30] PCI-SIG. PCI Express Architecture. <http://www.pcisig.com>.
- [31] PCI-SIG. PCI I/O Virtualization. <http://www.pcisig.com/newsroom/news/press.releases/2005.06.06>.
- [32] I. Pratt. Xen Virtualization. Linux World 2005 Virtualization BOF Presentation.
- [33] Ian Pratt and Keir Fraser. Arsenic: A User-Accessible Gigabit Ethernet Interface. In *INFOCOM*, pages 67–76, 2001.
- [34] Quadrics, Ltd. QsNet. <http://www.quadrics.com>.
- [35] M. Rosenblum and T. Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *IEEE Computer*, pages 39–47, May 2005.
- [36] Marc Snir, Steve Otto, Steve Huss-Lederman, David Walker, and Jack Dongarra. *MPI—The Complete Reference. Volume 1 - The MPI-1 Core, 2nd edition*. The MIT Press, 1998.
- [37] J. Sugerman, G. Venkitachalam, and B. H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of USENIX*, 2001.
- [38] Mellanox Technologies. I/O Virtualization with InfiniBand. <http://www.xensource.com/company/xensummit.html/Xen.Virtualization.InfiniBand.Mellanox.MKagan.pdf>.
- [39] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. In *Proceedings of the 12th International Parallel Processing Symposium*, 1998.
- [40] University of Cambridge. Xen Interface Manual.
- [41] Voltaire. Fast I/O for Xen using RDMA Technologies. <http://www.xensource.com/company/xensummit.html/Xen.RDMA.Voltaire.YHaviv.pdf>.

- [42] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *ACM Symposium on Operating Systems Principles*, 1995.
- [43] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer. Active Messages: A Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, pages 256–266, 1992.
- [44] C. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.
- [45] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Technical report, University of Washington, February 2002.
- [46] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of 5th USENIX OSDI*, Boston, MA, Dec 2002.

This is the html version of the file <ftp://ftp.cse.ohio-state.edu/pub/tech-report/2005/TR63.pdf>.
 Google automatically generates html versions of documents as we crawl the web.
 To link to or bookmark this page, use the following url: http://www.google.com/search?q=cache:F_D0uIGZ6a4J:ftp://ftp.cse.ohio-state.edu/pub/tech-report/2005/TR63.pdf+tr63.pdf+Ohio+State&hl=en&ct=clink&cd=1&gl=us

Google is neither affiliated with the authors of this page nor responsible for its content.

These search terms have been highlighted: **ohio state**
 These terms only appear in links pointing to this page: **tr63 pdf**

Page 1

InfiniBand Support in Xen Virtual Machine Environment

WEI HUANG, JIUXING LIU†, BULENT ABALI† AND DHABALESWAR K. PANDA

Computer Science and Engineering
 The Ohio State University
 Columbus, OH 43210
 {huanwei, panda}@cse.ohio-state.edu

† IBM T. J. Watson Research Center
 19 Skyline Drive
 Hawthorne, NY 10532
 {jl, abali}@us.ibm.com

Technical Report
 OSU-CISRC-10/05-TR63

http://64.233.167.104/search?q=cache:F_D0uIGZ6a4J:ftp://ftp.cse.ohio-state.edu/pub/tech... 6/15/2007

InfiniBand Support in Xen Virtual Machine Environment

Wei Huang Jiuxing Liu † Bulent Abali † Dhabaleswar K. Panda
Computer Science and Engineering † IBM T. J. Watson Research Center
The Ohio State University 19 Skyline Drive
Columbus, OH 43210 Hawthorne, NY 10532
{huanwei, panda}@cse.ohio-state.edu {jl, abali}@us.ibm.com

Abstract

Virtual machine (VM) technologies provide the benefit of running multiple OS instances in a single physical machine, as well as other advantages such as performance isolation and the ease of management. Meanwhile, InfiniBand is becoming a strong player in the area of data center and High Performance Computing (HPC) due to its high performance and the features such as Remote Direct Memory Access (RDMA) and user space communication. Thus, supporting InfiniBand in a virtualized environment has become increasingly important. However, current device virtualization models require the involvement of a hypervisor and/or a privileged virtual machine

able to achieve nearly the same raw performance as OpenIB Gen2 driver running in native Linux.

1 Introduction

With modern computers becoming increasingly powerful, virtual machine (VM) technologies are becoming more and more attractive to both the industry and the research communities. VM technologies allow many different virtual machines running in a single physical box, with each virtual machine possibly running a different operating system. VMs can also provide secure and portable environments to meet the demanding requirements of computing resources in modern computing systems [2].

(device domain), which may turn out to be a bottleneck for high performance communications.

In this paper, we present a novel design of Xen-IB, an InfiniBand device driver for a popular virtual machine monitor, Xen. We use a device domain bypass approach to avoid the involvement of the hypervisor and the device domain for time critical operations. Xen-IB has been implemented for Xen-Linux 2.6.12 in Xen 3.0. Our performance measurements show that the Xen-IB was

— This research was initiated during Wei Huang's visit to IBM T. J. Watson Research Center as a summer intern. It is supported in part by Department of Energy's Grant #DE-FC02-01ER25506 and National Science Foundation grants #CNS-0403342 and #CCR-0509452.

Device I/O is handled in different ways in various VM technologies. For instance, in VMware Workstation [14], which provides full virtualization of the x86 architecture, device I/O relies on user level emulation or switching back to the host operating system. VMware ESX Server makes direct accesses to high performance I/O devices from the hypervisor [15]. In Xen [4], a high performance virtual machine monitor originally developed at the University of Cambridge, device I/O follows a split-driver model. Only an isolated device domain (IDD) has access to the hardware using native device drivers. All other virtual machines (guest domains) need to pass the I/O re-

1

Page 3

quests to the IDD to access those devices. This control transfer between domains needs the involvement of the hypervisor and therefore has a performance penalty compared to native device operations.

The InfiniBand Architecture (IBA) [6] defines a System Area Network (SAN) for interconnecting processing nodes and I/O nodes. It provides memory semantics (RDMA) as well as channel semantics (send/receive). InfiniBand provides OS-bypass communication schemes. Many time critical communication operations can be performed directly from the user space without the overhead of going through operating systems. InfiniBand provides very low latency and high bandwidth, making it a strong player in the field of high performance cluster computing and I/O.

Supporting high performance communication devices such as InfiniBand in VMs is important in many aspects. First, VMs like Xen are becoming increasingly popular on modern servers and workstations. Many of these powerful computers are equipped with high speed interconnects like InfiniBand. As a result, providing efficient

(Xen and VMware workstation), making it difficult to support OS-bypass features of InfiniBand. To address this issue, we expanded the split device driver model used in Xen by adding IDD-bypass features, which allow time critical operations to be carried out directly within guest domains. With this optimization, we remove the bottleneck of going through the device domain and greatly improve the performance.

The main contributions of our work are:

- We designed and implemented an InfiniBand driver for Xen guest operating systems based on OpenIB Gen2 stack [10]. We keep the same interfaces as provided by Gen2, so existing programs written using the Gen2 verbs interface are transparently supported.
- We enhanced our driver with the device domain (IDD) bypass approach. For time critical operations, Xen guest applications are able to bypass the device domain and directly communicate with the IB adapters without violating the Xen security model.

• We evaluated the performance of Xen-IB.

VM communication and I/O using these high speed interconnects is critical to fully achieve the benefit of virtualization. Further, InfiniBand is widely used in High Performance Computing (HPC) [13]. VM technologies may provide many valuable features for HPC, such as better checkpointing support, QoS and better cluster management. Although VMs currently are not widely used in HPC because of the overhead caused by virtualization, recently introduced high performance VMs like Xen, combined with a highly efficient InfiniBand support as demonstrated in this paper, can make VM based HPC a viable solution.

In this paper, we present a novel design of Xen-IB, an InfiniBand driver in the Xen virtual machine environment. Virtualizing an InfiniBand device poses unique challenges compared to other devices. Current device virtualization models in VMs require the involvement of either the hypervisor (VMware ESX) or the privileged domain

- we evaluated the performance of our solution with a set of micro-benchmarks. The evaluation showed that our Xen-IB implementation provides similar performance compared to OpenIB driver running on native Linux systems.

The rest of the paper is organized as follows: In Section 2, we present background information of the Xen virtual machine monitor and the InfiniBand architecture. In Section 3, we discuss detailed design and implementation issues for the Xen-IB driver. Performance evaluation results are given in Section 4. We discuss related work in Section 5 and conclude the paper in Section 6.

2 Background

2.1 Overview of the Xen Virtual Machine Monitor

In this section we give a brief overview of the Xen virtual machine monitor (hypervisor). We

2

also describe Xen inter-domain communication and its split device driver model.

Xen uses paravirtualization [18], in which host operating systems need to be explicitly ported to the Xen architecture. This architecture is similar to native hardware such as x86 architecture, with only slight modifications to support efficient virtualization. Since Xen does not require changes to the application binary interface (ABI), existing applications can run without any modifications.

VM0 (Domain0)	VM1 (Guest Domain)	VM2 (Guest Domain)
Device Manager and Control Software	Unmodified User Software	Unmodified User Software
Guest OS (XenLinux)	Guest OS (XenLinux)	Guest OS (XenLinux)
Back-end driver native	front-end driver	front-end driver

hosts a guest operating system. Instead of running in ring 0, Guest OSes are modified to run in ring 1, which prevents them from directly executing the privileged processor instructions. Guest applications, like on normal x86 machines, run in ring 3 (the least privileged level). Only domain0, which is created at the boot time, is allowed to access the control interface provided by the hypervisor. The guest OS on domain0 hosts the application-level management software and perform the tasks to create, terminate or migrate other domains through the control interface provided by hypervisor.

There is no guarantee that a domain will get a continuous stretch of physical memory to run a guest OS. So Xen makes a distinction between *machine memory* and *pseudo-physical memory*. Machine memory refers to the physical memory

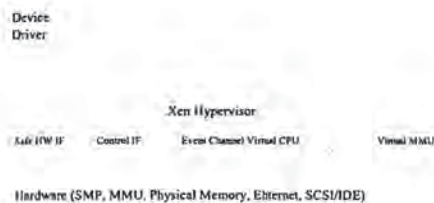


Figure 1. The structure of the Xen hypervisor, hosting three xenoLinux operating systems (courtesy [11])

Figure 1 illustrates the architecture of a physical machine running the Xen hypervisor. The Xen hypervisor is at the lowest level and has direct access to the hardware. Xen needs to ensure that the hypervisor, instead of the guest operating systems, is running in the most privileged processor-level. The x86 privilege levels are described by “rings”, from ring 0 (the most privileged) to ring 3 (the least privileged). The hypervisor is running in ring 0. It provides basic control interfaces needed to perform complex policy decisions in Xen architecture. Above the hypervisor are the Xen domains. There can be many domains running simultaneously. Each domain

installed in the machine. On the other hand, pseudo-physical memory is a per-domain abstraction, allowing a guest OS to treat its memory as consisting of a contiguous range of physical pages. Xen maintains the mapping between the machine memory and the pseudo physical memory. Only a certain specialized parts of the operating system needs to understand the difference between these two abstractions. Guest OSes allocate and manage their own hardware page tables, with minimal involvement of the Xen hypervisor to ensure safety and isolation.

In Xen, domains communicate with each other through *event channels*. Event channels provide an asynchronous notification mechanism between Xen domains. Each domain has a set of end-points (or ports) which may be bounded to an event source (e.g. a physical IRQ, a virtual IRQ, or a port in another domain) [17]. When a pair of end-points in two different domains are bound together, a “send” operation on one side will cause an event to be received by the destination domain. Event channels are only intended for sending notifications between domains. So if a domain wants to send data to another, the typical scheme is for a source domain to grant access to local mem-

3

ory pages to the destination domain. Then, these shared pages are used to transfer data.

Virtual machines in Xen usually do not have direct access to hardware. Since most existing device drivers assume they have complete control of the device, there cannot be multiple instantiations of such drivers in different domains for a single device. To ensure manageability and safe access, device virtualization in Xen follows a split device driver model [5]. Each device driver is expected to run in an *isolated device domain (IDD)*, which also hosts a *back-end driver*, running as a dae-

Once submitted, a Work Queue Request becomes a *Work Queue Element (WQE)*, and are executed by Channel Adapters. The completion of work queue elements is reported through *Completion*

Queues (CQ). Once a work queue element is finished, a *Completion Queue Entry (CQE)* is placed in the associated completion queue. A kernel application can subscribe for notifications from HCA and register a callback handler with CQ. Completion queue can also be accessed through polling to reduce latency.

Initiating data transfer (posting work requests)

mon and serving the access requests from guest domains. Each guest OS uses a *front-end* driver to communicate with the back-end. The front-end and back-end drivers communicate through the shared memory page and the event channel mechanism described above. The split driver organization provides security: misbehaving code in a guest domain will not result in failure of other guest domains. To improve the throughput for devices that need large data transfer such as network I/O, only the request descriptors can be passed to the IDD. The actual data payload may be directly DMAed from the guest domain to the device. Since the split device driver model requires the development of front-end and back-end drivers for each individual device, not all devices are supported in guest domains right now.

2.2 InfiniBand Architecture

The InfiniBand communication stack consists of many layers. The interface presented by Channel adapters to consumers belongs to the transport layer. A queue-based model is used in this interface. A *Queue Pair (QP)* in InfiniBand Architecture consists of a send queue and a receive queue. The send queue holds instructions to transmit data and the receive queue holds instructions that describe where received data is to be placed. Communication instructions are described in *Work Queue Requests (WQR)*, or descriptors, and are submitted to the work queue.

and completion of work requests notification (poll for completion) are time-critical tasks that need to be performed by the application. In the Mellanox [8] approach, which represents a typical implementation of InfiniBand specification, these operations are done by ringing a doorbell. Doorbells are rung by writing to the registers that form the *User Access Region (UAR)*. UAR is memory-mapped directly from a physical address space that is provided by HCA. It allows access to HCA resources from privileged as well as unprivileged mode. Each UAR is a 4k page. Mellanox HCAs replicate the UARs up to 16M times. Each process using the HCA, no matter in kernel or user space, is allocated one UAR, which is remapped to the process's virtual address space. Posting a work request includes putting the descriptors (WQR) to QP buffer and writing the doorbell to the UAR. This process can be completed without the involvement of the operating system. Therefore it is very fast. CQ buffers, where the CQEs are located, can also be directly accessed from the process virtual address space. These OS-bypass features make it possible for InfiniBand to provide very low communication latency.

Mellanox HCAs require all buffers involved in communication be registered before they can be used in data transfers. The purpose of registration is two-fold: first an HCA needs to keep an entry in the Translation and Protection Table (TPT) so that it can perform virtual-to-physical translation and protection checks during data transfer;

4

second the memory buffer needs to be pinned in memory so that HCA can DMA directly into the target buffer, always target to the correct position. Upon the success of registration, a local key and a remote key are returned. They will be used later for local and remote (RDMA) accesses. QP and CQ buffers described above are just examples

3.1 Overview

Like many other device drivers, InfiniBand drivers cannot have multiple instantiations for a single HCA. Thus a split driver model approach is required to share a single HCA among multiple Xen domains

and CQ buffers described above are just normal buffers that are directly allocated from process virtual memory space and registered with HCA.



Figure 2. Architectural overview of OpenIB Gen2 stack

There are two popular stacks for InfiniBand drivers. VAPI [9] is the Mellanox implementation and OpenIB Gen2 [10] recently comes out as a new generation of IB stack provided by the OpenIB community. In this paper we implement InfiniBand drivers based on Gen2. Figure 2 represents the architecture of Gen2 stack.

3 Design and Implementation

In this section we present the design and implementation of Xen-IB, our InfiniBand driver for Xen. First we introduce a design that follows the traditional Xen split device driver model and the details of its major functional modules. Then we optimize the driver by directly accessing the HCA from guest domains for time critical tasks.

FIGURE 3

Figure 3 illustrates a basic design of our Xen-IB driver. The back-end runs as a daemon on top of the native InfiniBand driver in the isolated device domain (IDD). It waits for incoming connections from the front-end drivers in the guest domains. The front-end driver replaces the kernel HCA driver in normal OpenIB Gen2 stack. Thus the higher layer InfiniBand services can be supported smoothly in this design. Once the front-end driver is loaded, it establishes two event channels with the back-end daemon. The first channel, together with the shared memory page scheme described in Section 2.1, forms a device channel [5] which is used to process the requests initiated from the guest domain. The second channel is used for sending InfiniBand CQ and asynchronous events to the guest domain and will be discussed in detail later.



Figure 3. The Xen-IB driver structure with the split driver model

The Xen-IB front-end driver provides the same set of interfaces as normal Gen2 stack for kernel modules. It is a relatively thin layer whose tasks

include packing the command together with re-

conducted in IDD to ensure that all the references

include packing the command together with necessary parameters and sending it to the back-end through the device channel. The back-end driver re-constructs the commands, performs the operations with the native kernel HCA driver on behalf of the guest domain, and returns the result to the front-end driver. The back-end manages all InfiniBand resources on behalf of the guest domains. So all resources including QP/CQ buffers are actually allocated within the device domain. Only the handles to those resources are passed back to the front-end driver for later references.

The split device driver model in Xen poses difficulties for user-level direct HCA access in Xen guest domains. Since all resources are managed inside the device domain, the user-level HCA driver does not have direct access to the UARs as well as the QP/CQ buffers. It has to be changed so that the original OS-bypass operations should now also go through the kernel. To achieve this, we need to modify the user-level InfiniBand service module and the user access control part of the InfiniBand core modules.

3.2 Design Details

In the following, we first discuss in general how we support all InfiniBand operations. Then we explain in detail the implementation of major functional modules, including queue pair access, completion polling, memory registration and event handling.

Before applications can send messages using InfiniBand, it must finish several preparation steps including opening HCA, creating CQ, creating QP, and modifying QP status, etc. Those operations are usually not in the time critical path of the applications and can be implemented in a straightforward way. Basically, the guest domains forward these commands to the device domain (IDD) and wait for the acknowledgments after the operations are completed in IDD. All the resources are managed in IDD and the front-ends refer to these resources by handles. Validation checks must be

conducted in IDD to ensure that all the references are legal.

Next we look at some key functional modules of our Xen-IB driver:

- **QP Access:** QP accesses (posting descriptors) include writing the WQEs to the QP buffer and ringing the doorbell to notify the HCA. Then the HCA will use DMA to transfer the WQEs to internal HCA memory and perform the send/receive or RDMA operations. Since all resources are located in the back-end driver, including the QP buffers and the UARs, it is required that the doorbell is rung at the device domain, and the WQEs also have to be accessible by the back-end only. Thus, once a post operation occurs, the front-end driver will pack and send to the back-end driver the content of work request and the lists of the buffers that need to be transferred. The back-end takes the responsibility to reconstruct the work requests and post them through the native HCA driver. Note that in this process, we only send the DMA addresses of the buffers. The actual data in the guest domains will be directly accessed by the HCA using DMA.

The biggest problem for this approach, however, is its inefficiency. Involving the device domain can be very costly in terms of performance. And the need to change user-level access modules to support user-level verbs also makes implementation quite complicated.

Meanwhile there are advantages of this approach. In Xen, all the guest domains share the same HCA in the physical box. So it will be beneficial if we can have global information of how the HCA is being used. By forcing every post to go into the device domain, the back-end driver may put the work requests from the guest domains into different request queues, which can be used to

build different kinds of scheduling schemes for better QoS.

- **CQ Polling:** Once the work request is completed, HCA will put a completion entry (CQE) in the CQ buffer. However, since CQ buffers are allocated in the device domain, the front-end driver cannot directly poll the CQ buffer in the guest domains. Therefore, every time the application tries to poll a CQ, the front-end driver has to send a request to the back-end driver. The back-end driver will poll the CQ. And if there is any CQE newly generated, it sends the CQE content back to the front-end driver.

Polling CQ is a quite frequently used verbs operation and usually lies in the critical path of the communication. Since it is not efficient to go to the back-end in every polling, we should optimize it as much as possible. One optimization to improve the efficiency is CQE buffering. Even though the front-end driver may only ask for a few number of WQEs in a poll CQ request, the back-end driver returns all the CQEs currently in the CQ buffer. The front-end then returns to the application the needed CQEs and buffers the rest of them. The next time the application polls CQ, the front-end driver does not need to communicate with the back-end if the needed CQEs have already been buffered locally. In this way we can amortize the cost of visiting the back-end driver.

- **Memory Registration:** The InfiniBand specification requires that all the memory regions involved in data transfers be registered with HCA. With Xen's paravirtualization approach, all domains see the same DMA address as the real machine address. So there will be no extra address translation needed here. The information needed by memory registration is a list of DMA ad-

of the buffers, access flags and the virtual address that the application will use when accessing the buffer. Again, the registration happens at the device domain. The front-end driver simply needs to send above information to the back-end driver and get back the local and remote keys. Note that since the Translation and Protection Table (TPT) on HCA is indexed by keys, we don't need to worry if multiple guest domains try to register with the same virtual address.

For security reasons, the back-end driver can verify if the front-end driver offers valid DMA addresses belonging to the specific domain that it is running in. This check will make sure that all later communication activities of guest domains are within the valid address spaces.

- **Event Handling:** InfiniBand supports several kinds of CQ and QP events. The most commonly used is the completion event, which is described in section 2.2. Event handlers are associated with CQs or QPs when they are created. The application can subscribe for event notification by writing the appropriate command to the UAR page. When those subscribed events happen, the HCA driver will first get notified from HCA and then dispatch the event to different CQs or QPs according to the event type. Then the application that owns the CQ/QP will get a callback on their event handlers.

For Xen-IB driver, event handling is a bit more complicated than other operations. Events are generated for the device domain, since all QPs and CQs are actually created there. But the device domain cannot directly give a callback on the event handlers in the guest domains. To address this issue, we create a dedicated event channel between the front-end and the back-end driver. The back-end driver associates a special event han-

addresses that describes the physical locations

abler to each CQ/QP created due to requests

7

Page 9

from guest domains. Each time the HCA generates an event to these CQs/QPs, this special event handler gets executed and forwards the information such as the event type and the CQ/QP identifier to the guest domain through the event channel.

The front-end driver binds an event dispatcher as a callback handler to one end of the event channel after the channel is created. The event handlers given by the applications are associated to the CQs or QPs after they are successfully created. Once the event dispatcher gets an event notification from the back-end driver, it checks the identifier and gives the corresponding CQ/QP a callback on the associated handler. In this process, the front-end driver has to maintain a translation table between the identifiers and the actual CQ/QPs.

3.3 An IDD-Bypass Approach for Improving Performance

The Xen-IB driver follows a traditional Xen split device driver model. Every HCA resource including QP buffers, CQ buffers, and User Access Regions (UAR), is allocated in the isolated device domain (IDD). So every operation including time critical ones have to go to IDD. As a result, there can be significant performance loss due to the context switch and inter-domain communication. For example, in the native IB drivers, applications can directly post or poll completion even from the user space. But now as shown in Fig. 4, since all access to UARs have to go to the device domain, all those operations have to go through the kernel of the guest domain, the kernel



Figure 4. Working flow of the Basic Xen-IB driver



of the device domain, and comes all the way back, which adds large overhead to the latency.

Similar to the OS-bypass concept in InfiniBand user level access, we enhanced our driver by an IDD-bypass scheme that allows the time critical operations to be directly issued from guest domains. This poses two requirements on our design

8

Figure 5. Working flow of the IDD-bypass Xen-IB driver

Page 10

of Xen-IB driver: first, the UAR page must be accessible from the guest domain; second, both the QP and CQ buffers should be directly visible in the guest domain. Thus we changed the design of the front-end and the back-end drivers as follows:

- When a front-end driver is loaded, the back-end driver allocates a UAR and returns its page frame number (machine address) to the front-end. The front-end driver then remaps this page to its own address space so that it can directly access the UAR in the guest domain to serve requests from the kernel applications. In the same way, when a user application starts, the front-end driver also applies a UAR page from the back-end and remaps the page to the application's virtual memory space, which can be later accessed directly from the user space. Since all UARs are managed in a centralized manner in the IDD, there will be no conflicts between UARs in different guest domains.
- Creating CQs/QPs is no longer simply sending the commands to the back-end. Instead, it is divided into two stages. In the first stage, QP or CQ buffers are allocated in the guest domains and registered through the IDD. During the second stage, the front-end sends the CQ/QP creation commands to the IDD along with the keys returned from the registration stage to complete the creation

All other operations, such as creating and destroying QPs, still have to go through the device domain. But since usually these are not in critical path of communication, it has almost no impact on performance. We show the new stack of our Xen-IB driver in Figure 6.



Figure 6. IDD bypass design of Xen-IB driver

Note that allowing the front-end driver to directly post requests does not mean that we are violating Xen rules to make validation checks. All QP/CQ creations and buffer registrations go through the back-end driver, which basically means all parts involved in the communication can be verified with the device domain.

process. Address translations are indexed by keys, so in later operations the HCA can directly read WQRs from and write the CQEs back to the buffers located in the guest domains.

Since we also allocate UARs to user space applications in guest domains, the user level InfiniBand library now keeps its OS-bypass feature. As shown in Fig. 5, the IDD-bypass scheme greatly reduces the communication latency.

The strategies for memory registration and event handling still fit in this IDD-bypass model.

4 Performance Evaluation

In this section, we evaluate the performance of our Xen-IB driver using a set of OpenIB Gen2 micro-benchmarks. We illustrate that the IDD-bypass scheme shows significant performance improvements over the basic design by using kernel-level RDMA latency and throughput tests. And with user-level micro-benchmarks, we show that the improved Xen-IB driver provides performance very close to native hardware, with only slight performance degradation for event handling and memory registration.

9

Page 11

4.1 Testing Setup

We use two IBM xSeries 360 servers as our testbed. Each of them is equipped with an Intel Xeon 3.2GHz CPU, 1 GB memory and a Mellanox MHXL-CF128-T PCI-X InfiniBand HCA. The servers are connected with no switches in between. One of the servers (server A) is running Xen hypervisor with 2.6 kernel. RedHat AS4 are running both in domain0 and guest domains. OpenIB Gen2 stack is installed in domain0 as the native IB driver. The other server (server B) is running RedHat AS4 with 2.6.13 kernel. The same version of OpenIB Gen2 is installed on server B. Test environment settings on server B is fixed. Server A may have different settings to evaluate different schemes.

This asymmetric testing setup represents a scenario where servers in Xen guest domains use InfiniBand as an I/O transport to access remote devices which are hosted on a remote node that runs an unhypermvised OS.

4.2 Benefits of the IDD-Bypass Driver

In this subsection, we compare the performance of the basic native driver design with the

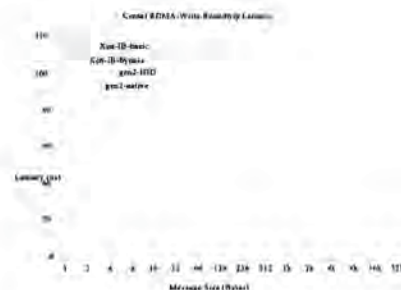
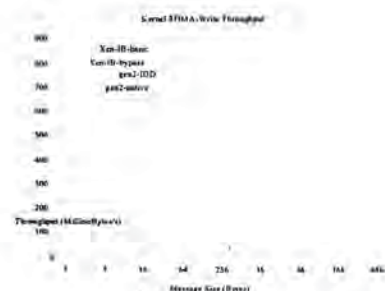


Figure 7. Kernel-level round-trip time for RDMA write operation



formance of the basic spin driver design with the IDD-bypass design using kernel Gen2 micro-benchmarks. We use kernel level RDMA latency and throughput tests to evaluate our designs.

In RDMA latency test, server A writes to server B using RDMA, while server B polls for the arrival of the data and replies back with a RDMA write of the same size. Since our test setup is asymmetric, we measure the round trip time of this rendezvous process instead of the one way latency. In RDMA throughput test, server A, which is running Xen, issues RDMA writes back to back to server B. And we measure the maximum throughput we can achieve. However, we do not measure the throughput from server B to server A because there is only DMA operations at the target side of RDMA, in which our Xen-IB driver is not involved.

Fig. 7 and Fig. 8 show the results for the

Figure 8. Kernel-level RDMA write Uni-directional throughput from server A to server B

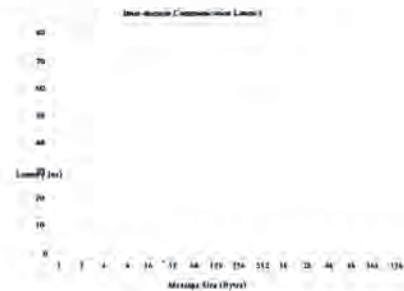


Figure 9. Inter-domain device channel one-way latency

10

RDMA latency and throughput tests. While the server B is always running native Linux, we compare the performance of kernel level RDMA write latency/throughput between the following four environments in server A: the basic Xen-IB driver in the guest domain (Xen-IB-basic), the IDD-bypass Xen-IB driver in the guest domain (Xen-IB-bypass), the native Gen2 driver in the device domain (gen2-IDD), and the native Gen2 driver on native Linux (gen2-native).

Note that the latency we show is the round-trip time. We can see that the basic Xen-IB driver has significantly higher latency than other drivers. The IDD-bypass driver, native driver in the device domain, and native driver on native Linux perform almost the same. We also show the one way latency of inter-domain communication on server A in Fig. 9, which explains the difference between the performance numbers. Going through the device domain adversely affect the throughput we

same performance as the native Gen2 driver in the device domain and the native Linux. This is not surprising because the critical path of these tests, posting descriptors and polling completions, is executed entirely in the guest domain, which can achieve performance close to native hardware. In the tests, we achieved 7.8 μ s round-trip latency (3.9 μ s one way) for 1 byte message and 830 MB/s throughput for 256k bytes message.



can achieve. As shown in Fig. 8, the peak value with the basic Xen-IB driver is only 770 Million Bytes/s at 128 KB while we can achieve around 830 MB/s with all other drivers. With each post costing at least 10 μ s (one way latency of inter-domain communication) to go to the device domain, it is almost impossible to post back-to-back sends to saturate the link unless the messages are sufficiently large.

4.3 Latency and Throughput

With kernel level benchmarks, we have already shown that involving the device domain in time critical operation adversely affects performance. So in the rest of this section, we focus on our IDD-bypass approach.

Fig. 10 and Fig. 11 show the latency and throughput for user-level micro-benchmarks. As in the kernel-level tests, the throughput test measures the uni-directional bandwidth from server A to server B.

As clearly shown in the figures, in both RDMA (indicated as rdma in figures) and send/receive (sr) tests, the IDD-bypass driver shows nearly the

Figure 10. User-level round-trip communication time

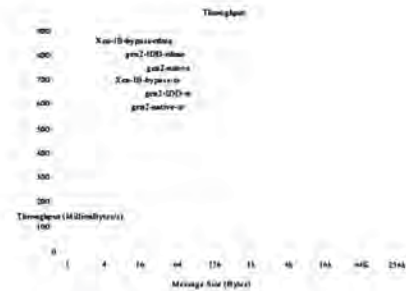


Figure 11. User-level Uni-directional throughput from server A to server B

4.4 Event Handling

The latency numbers we showed above are all based on polling schemes. In this section we show

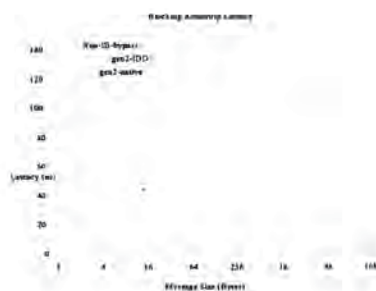


Figure 12. User-level send/rcv round-trip time using blocking verbs

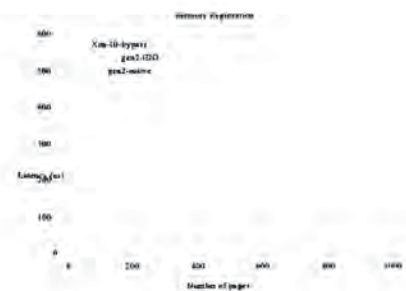


Figure 13. Memory registration time with respect to number of pages

the send/receive latency using blocking IB user-level verbs functions. This will reflect the efficiency of our event handling schemes.

Fig. 12 shows the send/receive round-trip latency using blocking verbs. The test is almost the same as send/receive ping pong test using polling. The only difference is that the process will block and wait for a completion event instead of busy polling on the completion queue. We show the results for server A running the IDD-bypass Xen-IB driver in the guest domain, the native IB driver in the device domain, and the native IB driver on native Linux. (Server B is running native Linux all the time.) Since Xen hypervisor is involved in the handling of device interrupts, round-trip latency is higher on Xen than native Linux, even with the native Gen2 driver. The extra cost of passing the event from the device domain to the guest domain adds another 2 to 10 μ s to latency.

4.5 Memory Registration

Memory registration is generally a costly operation. Fig. 13 shows the registration time of the IDD-bypass driver, the native Gen2 driver in the device domain and the native Gen2 driver on native Linux. The benchmark allocates a trunk of user buffers, then registers and unregisters the buffers multiple times and measures the average time for each registration.

As we can see from the graph, Xen adds consistently around 25%-35% overhead to the registration cost no matter how many pages we register. The Xen-IB driver in the guest domain adds more overhead, which increases with the number of pages involved in registration. This is because the more pages we register, the bigger the size of scatter gather list we need to send to the device domain through the inter-domain device channel. The performance drop is hard to avoid because every single page registered with the HCA should be verified by the device domain.

This observation indicates that if the registration is a time critical operation of an application, an efficient implementation of registration cache [16] becomes important.

5 Related Work

The concept of virtual machines was first introduced by IBM in 1960s to provide a time-sharing access environment to mainframe hardware [3]. It has been applied to operating systems for more than 30 years. Today there are numbers of solutions to virtualize x86 architecture. Denali [18] [19] uses para-virtualization as Xen does. VMware and Microsoft Virtual PC provide full x86 virtualization, which supports unmodified guest operating systems. Device accesses on full virtualization usually require that the hypervi-

12

sor intercept all I/O operations issued by the guest OS.

Recently both Intel and AMD provide hardware assistance to ease the full virtualization of x86 processor architecture. Intel VT-x technology [7] consists of a set of virtual-machine extensions (VMX) that support virtualization of processor hardware for multiple software environ-

handling and memory registration.

In the future, we plan to support the full set of InfiniBand services in Xen-IB, including management datagram (MAD) services. Meanwhile, since Xen is a fast developing open source project, we are continuously following Xen's newest features and integrate them into our work. We also plan to study the possibility to introduce

ments used by virtual machines. It will be supported in the next release of Xen [12]. Pacifica [1] is the AMD equivalent, which is claimed to be a functional superset of Intel VT. Though those hardware extensions make virtualization easier, the hypervisors still bear the responsibility for device I/O management.

There has been consistent effort to support high performance I/O devices in virtualized environment. Approaches used in Xen are discussed in Section 2. VMware workstation understands the semantics of I/O operations. Any accesses that actually interact with the physical hardware must go to the VMAp that runs in the host domain. Otherwise, the operations can be handled directly in the hypervisor. In VMware ESX Server, I/O devices are accessed from the hypervisor with the help of a global component that recognizes each individual virtual machine [15]. In either models, the I/O accesses need to involve the hypervisor, thus make it hard to support the OS-bypass features of InfiniBand.

6 Conclusions and Future Work

In this paper, we presented the design and implementation of Xen-IB, an InfiniBand driver for Xen virtual machine environment. We illustrated the general design of the driver and explained in detail all the key functional components. Our design bypasses the device domain for time critical tasks but still keeps every resource under the control of Xen. Our performance evaluations showed that the Xen-IB driver can provide performance close to native hardware under most circumstances, with expected degradation on event

VMs into high performance computing area. We will explore how to take advantages of Xen to provide better support of check-pointing, QoS and cluster management with minimum loss of computing power.

Acknowledgments

We would like to thank Charles Schulz, Dan Poff, Mohammad Banikazemi, and Scott Guthridge of IBM Research for valuable discussions and their support.

References

- [1] AMD. AMD Secure Virtual Machine Architecture Reference Manual, Rev. 3.01, May 2005.
- [2] P. M. Chen and B. D. Noble. When virtual is better than real. *Hot Topics in Operating Systems*, pages 133–138, 2001.
- [3] R. J. Creasy. The Origin of the VM/370 Time-sharing System. *IBM Journal of Research and Development*, 25(5):483–490, 1981.
- [4] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 164–177, October 2003.
- [5] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Reconstructing I/O. Technical Report UCAM-CL-TR-596, University of Cambridge, UK, August 2004.
- [6] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.2.
- [7] Intel Corporation. White paper: Enhanced Virtualization on Intel Architecture-based Servers.
- [8] Mellanox Technologies. <http://www.mellanox.com>.

- [10] Open InfiniBand Alliance. <http://www.openib.org>.
- [11] I. Pratt. Xen Virtualization. Linux World 2005 Virtualization BOF Presentation.
- [12] I. Pratt, K. Fraser, S. Hand, C. Limpach, and A. Warfield. Xen 3.0 and the Art of Virtualization. In *Proceedings of Linux Symposium*, 2005.
- [13] Top 500 Supercomputer Site. <http://www.top500.com>.
- [14] VMware Virtual Infrastructure Software. <http://www.vmware.com>.
- [15] J. Sugerman, G. Venkitachalam, and B. H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of USENIX*, 2001.
- [16] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. In *Proceedings of the 12th International Parallel Processing Symposium*, 1998.
- [17] University of Cambridge. Xen Interface Manual.
- [18] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Technical report, University of Washington, February 2002.
- [19] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of 5th USENIX OSDI, Boston, MA, Dec 2002*.

Evaluating the Impact of RDMA on Storage I/O over InfiniBand *

Jiuxing Liu

Dhabaleswar K. Panda

Mohammad Banikazemi^{††}

Computer and Information Science
The Ohio State University
Columbus, OH 43210
{liuj, panda}@cis.ohio-state.edu

[†]IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
mb@us.ibm.com

Abstract

Recently, several protocols such as iSER (iSCSI Extension for RDMA) and SCSI RDMA Protocol (SRP) have been proposed to improve the performance of storage I/O. These protocols promise a better performance by taking advantage of Remote Direct Memory Access (RDMA) operations. The need for design and development of new hardware and management infrastructure have made it difficult to evaluate these protocols and the impact of RDMA on storage I/O. On the other hand, recently InfiniBand solutions with RDMA capability have become available. Even though InfiniBand has been designed to be a unified interconnection network for both inter-processor and storage communication, its use has been mostly limited to inter-processor communication and not storage I/O. In this paper, we propose a new design through which the use of InfiniBand for storage I/O and the impact of RDMA operations on storage I/O can be evaluated.

Our design, which is called RDMA assisted iSCSI, extends the iSCSI protocol to exploit InfiniBand RDMA operations for data transfers. By keeping the TCP/IP based transport for management, control, and status messages unchanged, RDMA assisted iSCSI maintains compatibility with existing networking protocols and management infrastructure and makes its implementation in a reasonable time possible. We present the basic idea and several design and implementation issues and describe a prototype which has been implemented for the Linux operating system. Our evaluation demonstrates that RDMA can be very effective in improving the performance if certain requirements are met. We discuss what these requirements are. Our performance results show that with RDMA operations, we can improve the file read bandwidth from 96 MB/s to over 417 MB/s. We

also show latency and host CPU overhead can be reduced compared with the original iSCSI. Although our study is conducted in the context of InfiniBand, our results can be also applicable to other RDMA based designs such as iSER.

1 Introduction

During the last decade, the research and industry communities have been proposing and implementing user-level communication systems to address some of the problems associated with the legacy networking protocols. The Virtual Interface Architecture (VIA) [9] was proposed to standardize these efforts. More recently, InfiniBand Architecture [11] has been introduced which combines storage I/O with Inter-Processor Communication (IPC).

Due to its open standard and high performance, InfiniBand is gaining popularity in building high performance computing and database systems [12, 17]. These systems are cluster based, consisting of high-end machines connected by InfiniBand. To maintain compatibility and reduce management overhead, the TCP/IP protocol can be used in these systems for networking. IPoIB protocol [3] is the standard which defines the mapping of IP to InfiniBand. To achieve maximum performance, InfiniBand native protocols [7, 1] can be used for the communication needs of computing and database applications.

One crucial aspect of achieving high performance in cluster based systems is storage access. Many applications routinely process a large amount of data. Based on the interface provided, storage systems can be classified as Network Attached Storage (NAS) or Storage Area Networks (SANs). The NAS based approach has been discussed elsewhere [6, 8, 13]. In this paper, we focus on using the SAN based approach in InfiniBand clusters.

Traditionally, Fibre Channel (FC) has been used to build high performance SANs. However, in order to use FC in an

* This research is supported in part by Department of Energy's Grant #DE-FC02-01ER25506 and National Science Foundation's grants #CCR-0204429 and #CCR-0311542.

[†] Corresponding author.

InfiniBand cluster, a separate FC network must be installed. For large clusters, the cost of additional hardware can be prohibitive. Therefore, it is desirable to build a SAN with existing hardware in an InfiniBand cluster. To address this issue, several solutions have been proposed.

SCSI RDMA Protocol (SRP) [23] is an ANSI standard which takes advantages of InfiniBand Remote Direct Memory Access (RDMA) to carry SCSI traffic. SRP can be implemented directly over native InfiniBand interfaces and achieve very high performance. However, SRP has to be implemented from scratch and it requires separate protocols and infrastructures to be developed for management.

Recently, iSCSI [26] has been proposed which uses TCP/IP as the underlying communication protocol for storage access. Since iSCSI is implemented on top of TCP/IP, usually no separate network is needed for storage. Using TCP/IP protocols also greatly simplifies storage management because the same infrastructure in TCP/IP networks can be used to manage storage networks. Since iSCSI already defines IP protocol over InfiniBand, it seems more natural to use iSCSI in an InfiniBand cluster to provide storage access. If the storage target does not have an InfiniBand interface, we can use gateways to connect it to the InfiniBand network. Then clients can use iSCSI to access storage through the gateway, as shown in Figure 1.

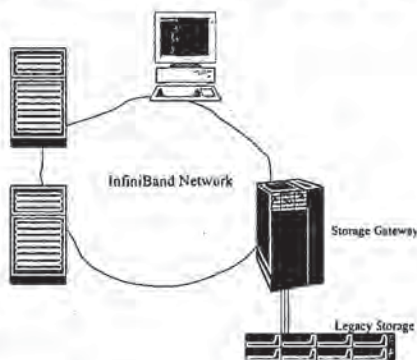


Figure 1. Using iSCSI in an InfiniBand Cluster

One problem in iSCSI is that overhead in TCP/IP may prevent one from achieving the highest storage performance. One major component of this overhead is due to extra memory copies, which greatly degrades storage bandwidth and increases host CPU utilization. To address this problem, special iSCSI customized HBA cards have been designed which can directly place data into destination buffers. However, this approach is not feasible with iSCSI running over InfiniBand without adding new hardware. As an extension to the original iSCSI protocol, iSER [19, 4] was proposed recently which exploits RDMA over IP [20]

to eliminate extra memory copies in the TCP/IP protocol. Similar to HBA based methods, this approach requires special "RDMA enabled NICs" (RNICs) and therefore is not feasible in an InfiniBand cluster either.

Although existing approaches cannot be used directly, the basic idea of placing data directly into the destination buffer without extra memory copies, can still be applicable in an InfiniBand cluster. Based on this idea, we propose using native RDMA operations in InfiniBand to enhance the original iSCSI protocol. Our design, which is called *RDMA assisted iSCSI*, preserves many advantages of the original iSCSI protocol such as backward compatibility and ease of management. The key idea of our design is to offload SCSI read and write data transfer in iSCSI from TCP/IP to InfiniBand RDMA operations. With this design, we not only avoid unnecessary copies, but also simplify the communication protocol. Although a similar idea was sketched elsewhere [27], in this paper we present a thorough study of this approach. We not only discuss our main idea as well as many design issues, but also carry out detailed performance evaluation. Our performance study shows that the RDMA assisted design can greatly improve storage bandwidth and reduce client overhead. It can achieve more than four times the bandwidth compared with the original iSCSI protocol in our InfiniBand cluster.

Our study in this paper not only provides a practical storage solution for InfiniBand clusters, but also can give insights into the potential of using RDMA in storage communication protocols in general. For example, iSER protocol requires RNICs that supports RDMA over IP. Although no such NICs are available yet, our design can serve as a close approximation to the iSER based approach. Therefore, some of our conclusions are also applicable to iSER.

In the remaining part of the paper, we first introduce some background information and related work in Section 2. In Section 3, we present the general idea of our RDMA assisted iSCSI design. We discuss detailed implementation issues in Section 4. Performance evaluation results are presented in Section 5. In Section 6, we give our conclusions and briefly mention some of the future work.

2 Background and Related Work

2.1 InfiniBand Architecture and iSCSI

The InfiniBand Architecture (IBA) [11] defines a switched network fabric for interconnecting processing nodes and I/O nodes. It provides a communication and management infrastructure for inter-processor communication and I/O. In an InfiniBand network, processing nodes and I/O nodes are connected to the fabric by Channel Adapters (CA). There are two kinds of channel adapters: Host Channel Adapter (HCA) and Target Channel Adapter (TCA).

HCAs are connected to processing nodes while TCAs connect devices. InfiniBand Architecture supports both channel and memory semantics. In channel semantics, send/receive operations are used for communication. In memory semantics, InfiniBand supports Remote Direct Memory Access (RDMA) operations, including RDMA write and RDMA read. RDMA operations are one-sided and transparent to the software layer at the receiver side.

iSCSI [26, 15] is a protocol that defines a transport layer for SCSI over TCP/IP. In this protocol, SCSI command, data and status are encapsulated into iSCSI Protocol Data Units (PDUs). iSCSI PDUs are then transferred between initiators and targets using the TCP/IP protocol. Since iSCSI is built on top of TCP/IP, it does not require a separate physical network for storage access traffic. Thus, its deployment and management cost is considerably less than the cost of other protocols such as Fibre Channel (FC). Different software/hardware combinations can be used to implement iSCSI [22, 21]. The iSCSI protocol can be handled completely in software. However, to improve storage access performance and reduce host overhead, the protocol can also be offloaded to the network interface cards [15, 16]. In the latter approach, it is possible to directly place SCSI data into the final destination memory without using any intermediate buffers in TCP/IP.

2.2 Related Work

The performance limitations of TCP/IP in high speed networks have been studied in [5]. In this subsection, we discuss the existing work which uses RDMA or similar approaches to address performance issues due to TCP/IP overhead in the iSCSI protocol. These include the iSCSI Extension for RDMA (iSER) protocol [19, 4], direct data placement in iSCSI Host Bus Adapters (HBAs) [21, 15], and the Voltaire iSCSI RDMA approach [27].

The iSER protocol was proposed recently to extend the iSCSI protocol by taking advantage of the iWARP [20] protocol suite. By using RDMA over IP provided by iWARP, iSER allows RDMA enabled NICs (RNICs) to directly put or get data from client buffers, without using intermediate memory copies in TCP/IP. This approach is similar to ours in that our design also uses RDMA to speed up iSCSI protocol and eliminate extra data copies. However, instead of using RDMA over IP and RNICs, which are not yet available, we have based our study on the RDMA operations provided by the InfiniBand Architecture and have shown how we can provide a high performance storage solution for InfiniBand based clusters. In many cases, our design can serve as an approximation of iSER. Thus, our study also provides insights into the potential of iSER.

The iSCSI protocol itself provides some general mechanisms to enable direct placement of SCSI data without extra

buffering in the TCP/IP protocol [15]. Using these mechanisms, a network interface card can be customized to allow iSCSI and TCP/IP to be offloaded. These iSCSI aware NICs are often called iSCSI Host Bus Adapters (HBAs). The iSCSI protocol provides information necessary for direct data placement. The HCAs then use this information to do placement and avoid extra copies. The basic idea of direct data placement in HBAs is similar to the aforementioned RDMA based iSER approach. However, instead of using a general RDMA interface, HBAs often rely on ad hoc protocols. Therefore, they tend to be incompatible with similar products. Because they do not have a uniform interface, it is difficult to develop software that works with different HBAs.

Recently, Voltaire Inc. has published a white paper describing how InfiniBand RDMA can be used in iSCSI for InfiniBand clusters [27]. Among all related work, their idea is most similar to ours. However, only high level concepts are described in their work. In this paper, we present many design issues and also conduct detailed performance evaluation.

3 Design of RDMA Assisted iSCSI over InfiniBand

As we have mentioned, using iSCSI in an InfiniBand cluster has the advantage of reusing existing software implementations and management infrastructures, which makes it possible to achieve fast deployment and easy transition. However, TCP/IP protocol over InfiniBand (IPoIB) [3] also has high overhead, making it difficult to obtain high performance and low overhead for storage access. Therefore, we base our design on the original iSCSI protocol and focus on investigating how InfiniBand features can be utilized to improve its performance.

The key idea in our design is to offload data transfers in iSCSI (SCSI read and SCSI write) from TCP/IP to InfiniBand RDMA operations. Figure 2 shows the bandwidth of raw InfiniBand RDMA and TCP over IPoIB in our testbed. (The details of the testbed used is described in Section 5.1.) We can see that for large messages, InfiniBand achieves much higher performance than IPoIB. This also suggests that TCP/IP over InfiniBand has much larger overhead. A large portion of this overhead is due to extra memory-to-memory copies in the TCP/IP protocol, which increases with the message size. In the iSCSI protocol, control traffic usually consists of small messages. However, data transfers (SCSI read and SCSI write) usually involve large messages. Therefore, in order to achieve high performance in the storage protocol, one has to reduce or eliminate this overhead in iSCSI data transfers. Next, we will discuss several design issues.

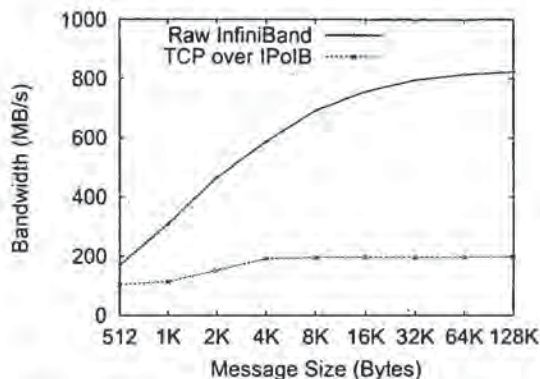


Figure 2. Bandwidth Comparison of Raw InfiniBand and TCP/IP over InfiniBand

3.1 Using RDMA for Data Transfer

In iSCSI, all communication traffic, including control information and data, goes through TCP/IP. Figure 3 shows the communication of SCSI read commands in the iSCSI protocol. To process a SCSI read command, iSCSI first creates a packet (or PDU) to transfer the read request from the initiator to the target. After getting this request, the target processes it and sends back the data to be read. Since iSCSI has a limit for the maximum number of bytes in a single PDU, more than one PDU may be required to send the data. Also the initiator may have limited buffering space. Therefore, after sending a certain amount of data, the target may need to wait for a flow control update from the initiator before it can continue sending more data. After all data has been sent, the target sends back the status information for the SCSI read command. For a SCSI write command, similar procedures happen, as shown in Figure 4.

The above approaches to handle read and write commands have potential performance problems when used directly in an InfiniBand cluster. First, as shown in Figure 2, TCP/IP over InfiniBand is not able to deliver high bandwidth. Therefore, performance for SCSI read and write will also be limited. Second, we see that for a single SCSI read or write command, multiple data packets and flow control packets may be required. Typically, each packet will generate at least one interrupt at the receiver. Since interrupts have high overhead, multiple incoming packets may significantly degrade performance. Also, having multiple packets means that the iSCSI protocol layer and the underlying communication layer must interact with each other multiple times to process these packets. These interactions can also increase communication overhead.

To address these problems, we study the use of RDMA

to offload SCSI reads and writes to InfiniBand RDMA operations. For SCSI read, instead of using TCP/IP, we let the target use RDMA write to transfer data directly to the destination buffer at the initiator. Similarly, in a SCSI write operation, the target uses RDMA read to get data from initiator. Figures 5 and 6 show the communication protocol after the offloading. RDMA operations require the user buffers to be registered. In our design, this registration information, together with buffer addresses, is attached with request messages from initiators.

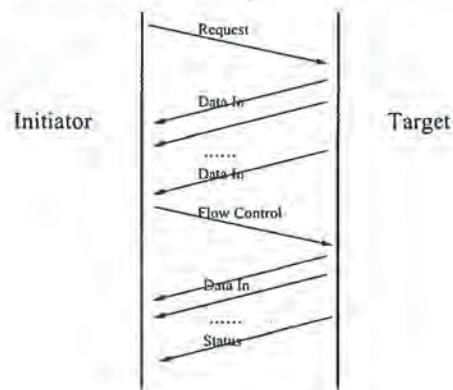


Figure 3. Data Read in iSCSI

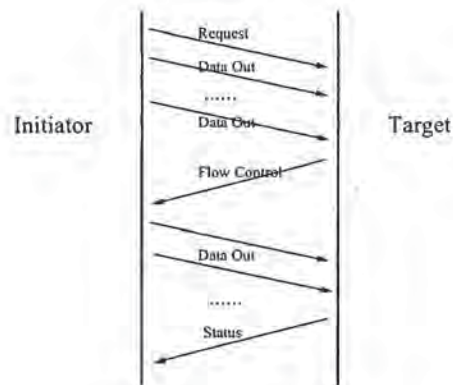


Figure 4. Data Write in iSCSI

We should note that control packets, including SCSI request and status packets, still go through TCP/IP in our design. Since control packets are usually small, communication performance is not significantly limited by the underlying TCP/IP protocol. More importantly, existing iSCSI implementations are usually driven by the processing of various control packets. Therefore, preserving the communication and processing of control packets can minimize port-

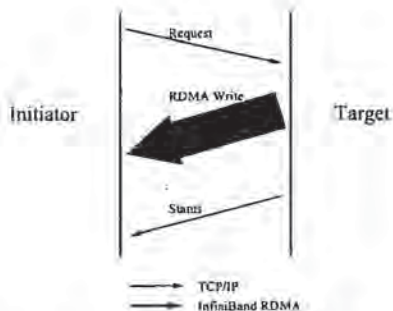


Figure 5. Data Read in iSCSI with RDMA

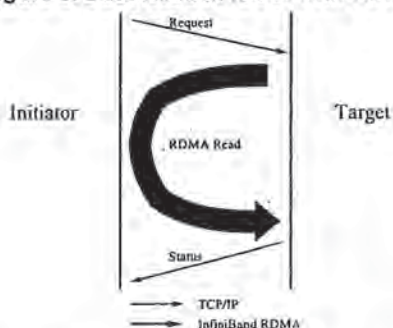


Figure 6. Data Write in iSCSI with RDMA

ing efforts from existing iSCSI implementations. We plan to study using InfiniBand also for control packets in future.

Using RDMA operations for iSCSI data transfer gives several advantages compared with the original iSCSI. First, since RDMA read and RDMA write are native InfiniBand operations, they have much higher performance than TCP/IP over InfiniBand which suffer from memory copies, as we have seen in Figure 2. Therefore, we can potentially achieve much higher bandwidth for data transfer in iSCSI. Second, since RDMA operations transfer data directly to or from the buffers at the initiator, there is usually no need for software flow control. If the buffers are contiguous, only one RDMA operation is needed for the entire SCSI read or SCSI write request. As a result, the overhead of processing multiple packets is also eliminated.

3.2 Dealing with Memory Registration Cost

By offloading SCSI data transfers to RDMA operations, RDMA assisted iSCSI can eliminate extra copies in the communication protocol. However, RDMA operations require that communication buffers are registered first. Buffer registration usually serves two purposes. First, it ensures the buffer will be pinned down in physical memory so that it can be safely accessed by InfiniBand hardware using DMA. Second, it provides the InfiniBand hardware with address translation information so that the buffer can be accessed

by virtual addresses. Buffer registration usually is a very expensive operation. Figure 7 shows the cost of registration for buffers of different sizes in our InfiniBand testbed. We can see that the cost is very high. Therefore, one central issue in our RDMA based design is to reduce or eliminate memory registration cost whenever possible. This problem can be easily addressed at the target side since storage targets usually have total control over their buffers and it is possible for them to pre-register the buffers to avoid registration cost. Next, we will discuss several techniques to deal with this issue at the initiator side.

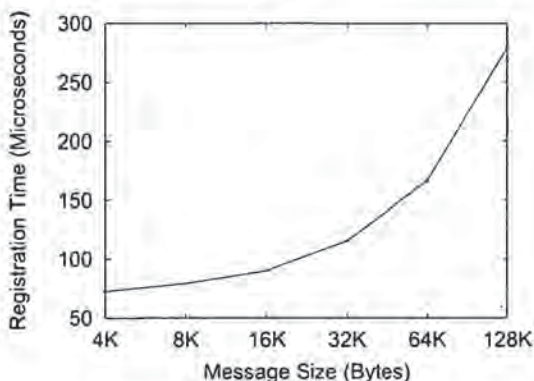


Figure 7. InfiniBand Memory Registration Cost

3.2.1 Memory Registration Cache (MRC)

One way to avoid memory registration cost is to use a memory registration cache. The basic idea is to maintain a cache of registered buffers and to do buffer de-registration in a lazy manner. When a buffer is first registered, it enters the cache. When it is de-registered, the actual de-registration is not carried out. Instead, the buffer stays in the cache. Therefore, when the buffer is used next time, it does not need to be registered since it is still in the registration cache. A buffer is de-registered when there are too many buffers in the registration cache.

The effectiveness of memory registration cache depends heavily on buffer reuse patterns. In MPI applications, we have observed high reuse rate of communication buffers [12]. Therefore, registration cache is very effective in that context. However, the buffer reuse pattern in storage systems may be quite different from those in scientific applications. Usually, data is cached at the client side, either by the operating system (file system cache) or the applications themselves (such as database systems). In these cases, after a I/O buffer is used, it will hold cached data. Therefore, it

is unlikely to be re-used again. As a result, buffer reuse rate may be low for storage clients. Therefore, memory registration cache may not be an effective technique for our design.

3.2.2 Fast Memory Registration (FMR)

Another way to reduce memory registration cost is to use a technique called Fast Memory Registration (FMR). In FMR, memory registration is divided into two steps. The first step is responsible for allocating resource needed by the registration, including address translation table in the InfiniBand HCA. The actual registration is done in the second step using resource obtained from the first step. Since resource allocation and de-allocation can be managed separately and in a batched fashion, usually only the second step will occur in the critical path of data transfer. As a result, FMR can achieve much faster memory registration.

FMR exists in RDMA over IP [20] and some InfiniBand implementations [14]. It is a very general technique and not dependent on application reuse patterns. Therefore, it can be used to reduce memory registration overhead in our design.

3.2.3 Zero-Cost Kernel Memory Registration (ZKMR)

In this subsection, we propose a new technique called Zero-Cost Kernel Memory Registration (ZKMR) to reduce memory registration cost for our storage clients. ZKMR is based on the observation that all buffers handed to our SCSI driver must have been pinned down in memory, otherwise the buffers would not be safely used for I/O operations. As we have discussed, memory registration include buffer pinning and address translation. Since buffer pinning is already done, we only need to inform InfiniBand hardware necessary address translation information.

We achieve the goal of address translation in two steps. First, we register all physical pages. In Linux, since all physical pages are mapped to a contiguous kernel virtual memory area, this step can be done with a single memory registration operation. Also since this memory area is physically contiguous, the registration consumes very little resource in the HCA. In the second step, we translate a buffer address to its physical address. Once this translation is done, registration information will be available because all physical pages are already registered in the first step.

We should note the first step in ZKMR is done during initialization. Thus, only the second step needs to be performed during memory registration. In Linux, this step can usually be achieved by a single arithmetic operation on the buffer address. We name it Zero-Copy Kernel Memory Registration because of its negligible cost. Although our implementation of ZKMR is specific to Linux, the general idea can be applied to other operating systems.

One drawback of ZKMR is that all physical pages must be registered. Since InfiniBand can only registered virtual addresses, this requires that all physical memory must be mapped to kernel virtual address space. In certain cases¹, not all physical pages can be mapped. In these cases, ZKMR cannot be used for memory pages that are not mapped initially.

3.3 Session Management

An iSCSI session is a logical association between the initiator and the target. A session must be established before iSCSI is functional. An iSCSI session usually consists of two phases: a login phase and a full-featured phase.

After the initiator has discovered an appropriate iSCSI target, It establishes a TCP/IP connection with the target. Then the login phase begins. One important task in the login phase is exchanging and negotiation of session parameters. After both sides agree on a set of parameters, the session enters full-featured phase, where the initiator can issue SCSI requests to the target.

Our design does not change the way bootstrapping and target discovery are done. Therefore, existing protocols such as iSNS [25] can still be used. The session login is also carried out as the original iSCSI protocol. However, we add a new session parameter which indicates whether RDMA offloading is supported or not. If both the initiator and the target are in the same InfiniBand cluster and support offloading, RDMA assisted iSCSI will be used to achieve better performance. Otherwise, RDMA offloading will not be used and the communication will be done as in the original iSCSI protocol. Therefore, our implementation has high compatibility with existing implementations. For example, it is able to inter-operate with existing iSCSI implementations on the same InfiniBand cluster that do not support offloading. It can also work with targets across a WAN that do not have InfiniBand connections.

3.4 Reliability and Security

The original iSCSI protocol puts much focus on reliability. The TCP/IP protocol uses checksum to help detect corrupted packets. In iSCSI, Cyclic Redundancy Code (CRC) can be used to achieve higher reliability. The use of CRC is optional and can be negotiated during login negotiation.

In our design, SCSI read and write data transfer uses InfiniBand RDMA operations whenever possible. Control traffic goes through TCP/IP, which also uses InfiniBand as the underlying data link layer. Since InfiniBand provides its own end-to-end CRC [11], CRC is no long necessary in the iSCSI protocol. As a result, iSCSI CRC is always negotiate to "none" when RDMA offloading over InfiniBand is used.

¹For example, Linux on X86 with more than 1GB physical memory.

Security is another important issue. During login phase, different protocols such as Challenge Handshake Authentication Protocol (CHAP) and Secure Remote Password (SRP) can be used for authentication in our design, just as in the original iSCSI protocol. To achieve data confidentiality, IPsec [24] can be used in the original iSCSI protocol. However, since SCSI read and write data transfer bypasses TCP/IP in our RDMA assisted iSCSI, IPsec cannot be used directly and confidentiality must be addressed explicitly. Since currently InfiniBand clusters usually consist of trusted nodes, most of the time performance is of higher priority than data confidentiality. However, we plan to investigate how to provide data confidentiality in future.

4 Implementation

We have implemented a prototype of our design. In this section, we provide an overview of our prototype, followed by discussions about issues such as memory registration, buffer re-organization and target implementation.

4.1 Implementation Overview

The Linux operating system organizes its SCSI subsystem in a layered structure [10, 18], as shown in Figure 8. At the upper level, Linux has different components to provide SCSI device abstractions such as block devices (SD for hard disks and SR for CDROMs) and character devices (ST for tape devices and SG for SCSI generic devices). Usually a SCSI hard disk is accessed through the SD interface. The middle layer is common for all devices. It is responsible for transforming commands from upper layer into SCSI requests. It then hands over these requests to lower layer drivers. At the bottom, lower layer drivers interface with Host Bus Adapters or provide an emulation layer for non-SCSI devices.

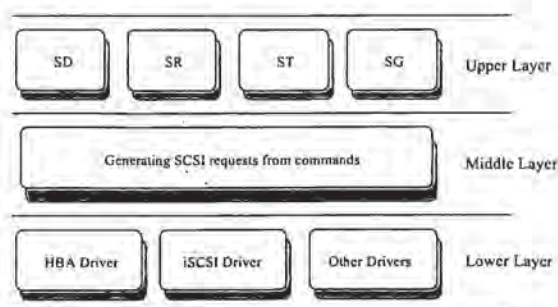


Figure 8. Linux SCSI Subsystem

At the client side, we have implemented our design as a SCSI driver at the lower layer of the Linux SCSI layers. Instead of driving a piece of hardware, our driver uses TCP/IP

and InfiniBand RDMA to deliver SCSI commands to a target node. As a result, storage provided by the target can be used transparently, just like any local SCSI disks. Our implementation is based on the Intel iSCSI project in SourceForge [2]. We used the InfiniBand Access Layer (IBAL) [1] to access functionalities provided by the underlying InfiniBand hardware.

4.2 Implementation of Memory Registration Handling

In the previous section we have proposed three techniques to reduce or avoid buffer registration cost for storage clients: MRC, FMR and ZKMR. As we have discussed, MRC may not be an effective method for our design. Currently, FMR is not yet support in IBAL. Therefore, we have used ZKMR in our implementation.

4.3 Buffer Re-Organization

In order to improve performance, another optimization in our design is to reduce the number of RDMA operations for each SCSI read or write request. For each request, the target side information includes a LUN, a start address and a size. The client information usually includes a list of buffers. In InfiniBand, each RDMA operation can only access one contiguous buffer at the remote side while the local buffers can be a list of buffers (gather/scatter list). To reduce the number of RDMA operations needed, we re-organize the client buffer list to find as many contiguous chunks as possible. For each chunk, we use only one RDMA operation for data transfer. One example of this process is illustrated in Figure 9. In this example, initially the client side has four buffers. However, we can combine them into a single buffer through re-organization.

Buffer reorganization is very effective when our RDMA assisted iSCSI is used for Linux file system I/O. Because of the way Linux manages its page cache, most of the time the re-organizing process is able to combine the client buffer list into a single buffer. Thus, the data transfer process can be achieved using a single RDMA operation with very low overhead.

4.4 Target Implementation

In our prototype, we implemented a RAM disk based storage target running in the user space. Since we are interested in the benefits of RDMA in iSCSI, using a RAM disk helps us to make sure physical disk access will not become the performance bottleneck. In future, we plan to extend our implementation to a disk based storage target which uses main memory as cache.

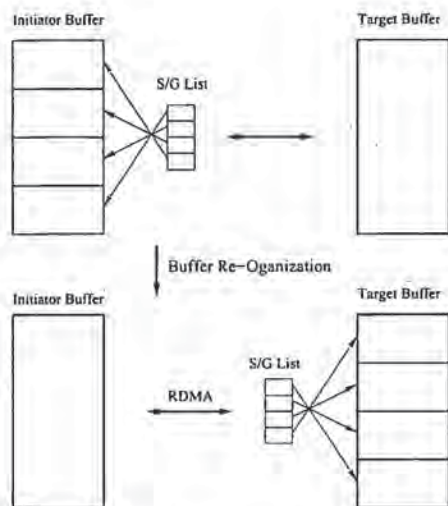


Figure 9. Client Based Buffer Reorganization

Our target implementation is single-threaded and serves one SCSI request at a time. Since most of the time client overhead is the performance bottleneck, currently this design is not a significant restriction. We plan to extend our implementation so that the storage target can handle multiple SCSI request simultaneously.

5 Performance Evaluation

In this section, we compare the performance of our RDMA assisted iSCSI with the original iSCSI implementation [2], which runs on top of iPoIB [3]. We also evaluate the impact of various techniques we used, such as Zero-Cost Kernel Memory Registration (ZKMR) and buffer reorganization.

5.1 Experimental Testbed

Our experimental testbed consists of a cluster system with a number of SuperMicro SUPER P4DL6 nodes. Each node has dual Intel Xeon 2.40 GHz processors with 512K L2 cache and a 400 MHz front side bus. The machines are connected by Mellanox InfiniHost MT23108 DualPort 4X HCA adapter through an InfiniScale MT43132 Eight 4x Port InfiniBand Switch. The HCA adapters work under the PCI-X 64-bit 133MHz interfaces. We used the Linux Red Hat 7.2 operating system with 2.4.18 kernel. SourceForge IB Access Layer version 1.118 was used for InfiniBand and iPoIB.

5.2 Benefits of RDMA Assisted Design

In order to study the benefits of our RDMA assisted design, we compare it with the original iSCSI implementation using micro-benchmarks such as bandwidth, latency and client host overhead. To avoid adding more complexity, we focus on SCSI read performance.

5.2.1 Bandwidth

In the bandwidth test, we use buffered read calls to get data from the iSCSI device file. Thus, the data will first go into the file system cache, and then be copied to the user buffer. It should be noted that buffered read bandwidth is affected by file system read-ahead policies. We set the maximum read-ahead window size to 255 in the tests. We study the effect of read-ahead window size in the next subsection.

Figure 10 shows the results for different block sizes. From the figure we see that without RDMA, iSCSI can only achieve bandwidth up to 96 MB/s. Its performance is limited by both the underlying iPoIB implementation and the extra memory copies from file system cache to the user buffer. However, with the help of RDMA, we can achieve bandwidth up to 417 MB/s for 16 KB blocks. Its performance is limited mostly by the memory copies from file system cache to the user buffer and other client side overhead such as protocol processing and interrupts. From the figure we can also see that when the block size becomes larger than 16 KB (8KB for the original iSCSI) bandwidth starts to decrease. This is due to processor cache effects.

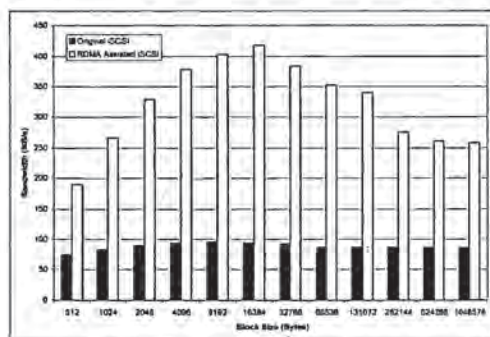


Figure 10. File Read Bandwidth

5.2.2 Impact of Read-Ahead Window Size

In the Linux file system, file read normally goes through file system cache. The operating system also has a mechanism to detect sequential file access patterns and uses file read-ahead (also called prefetching) to improve performance.

The read-ahead window size can have significant impact on performance.

In Figure 11, we show the bandwidth results for different maximum read-ahead window sizes. The block size used in the tests is 16KB. We can see that as the read-ahead window size increases, bandwidth also improves. This is because our tests read data sequentially and a bigger window size allows for more prefetching and more overlapping of different requests. For the original iSCSI, a window size of 15 is enough to achieve peak performance. However, RDMA assisted iSCSI needs a much larger window size (255) in order to achieve peak performance.

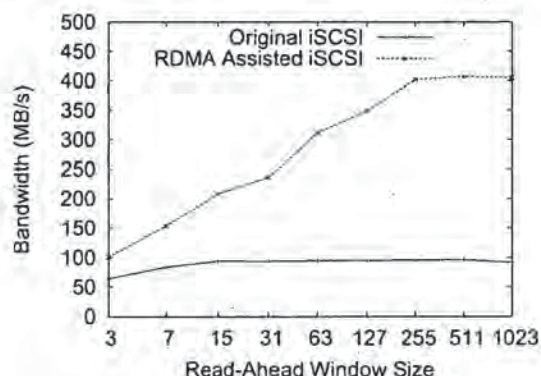


Figure 11. Impact of Read-Ahead Window Size

5.2.3 Latency

In the latency test, we need to eliminate the impact of file system read-ahead. Therefore, we used Linux raw I/O, which bypasses the file system cache and does not trigger file system read-ahead. Figure 12 shows the performance results. We can see that for block sizes larger than 1KB, RDMA assisted design outperforms the original iSCSI. However, factors of improvement are less compared with those in the bandwidth tests of the previous subsection. The reason is that requests for each block are serialized in the latency test. Since the RDMA assisted design uses TCP/IP for control messages, its performance is limited by the latency of IPoIB. In the bandwidth tests, different requests can be overlapped. Therefore, the RDMA assisted design can achieve a much better performance.

For small block sizes (1KB or less), we notice that the original iSCSI performs slightly better than the RDMA assisted design. This is due to an optimization in iSCSI called *phase collapse*. In phase collapse, the requested data and the status can be sent from the target to the initiator using a

single message, which can achieve better performance than using two separate messages. However, in the RDMA assisted design, two separate messages have to be sent: one through InfiniBand RDMA and one through TCP/IP. Therefore, we obtain a slightly lower performance.

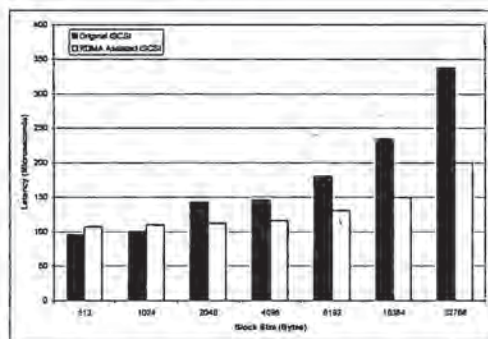


Figure 12. File Read Latency (Raw I/O)

5.2.4 Host Overhead

In the bandwidth tests which use buffered file system I/O, we have observed that client CPU utilization is close to 100% for both the original iSCSI and the RDMA assisted iSCSI. The high CPU utilization is mainly due to extra copies from file system cache to user buffers. This also implies that client side overhead is the performance bottleneck in the bandwidth tests. Since RDMA assisted iSCSI can achieve more than 4 times the bandwidth, it also means that with RDMA, CPU time can be reduced to less than one-fourth of the original time when transferring the same amount of data. Therefore, InfiniBand RDMA can greatly reduce host overhead.

We also show CPU utilization in the raw I/O latency tests. In Figure 13, we can see that RDMA assisted design consumes less CPU time compared with the original iSCSI for large block sizes. In fact, for large block sizes, CPU utilization decreases when the block size increases. However, CPU utilization for the original iSCSI increases with the block size. This is mostly due to the cost of copying. (For RDMA assisted design, there is a drop in CPU utilization for 1KB blocks. We are currently investigating this issue.)

5.3 Impact of Memory Registration and Buffer Re-Organization

In this section, we consider the impact of avoiding memory registration and doing buffer re-organization. As we have mentioned, we use ZKMR to reduce memory registration overhead. We also conducted bandwidth tests without using this technique. That is, we explicitly register and

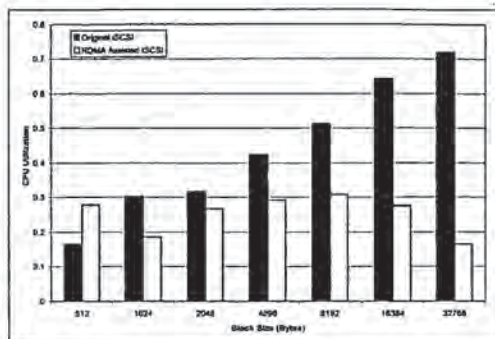


Figure 13. Host CPU Utilization in Raw I/O Latency Tests

de-register buffers during communication. The results are shown in Figure 14. We can see that performance drop significantly in this case. Therefore, avoiding high memory registration cost is very important in the RDMA assisted design.

In Figure 15, we compare bandwidth performance with and without using buffer re-organization. We can observe that using buffer re-organization only slightly improves the performance. This is because in the bandwidth tests, client side CPU is the bottleneck. Although buffer re-organization can reduce target side overhead, it does not bring much improvements to overall performance.

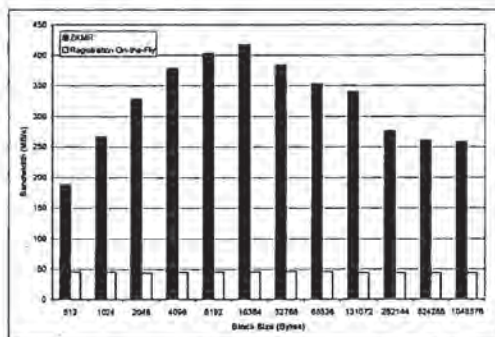


Figure 14. Impact of Memory Registration

6 Conclusions and Future Work

In this paper, we propose a design called RDMA assisted iSCSI that combines iSCSI with RDMA over InfiniBand which provides a high performance storage solution for In-

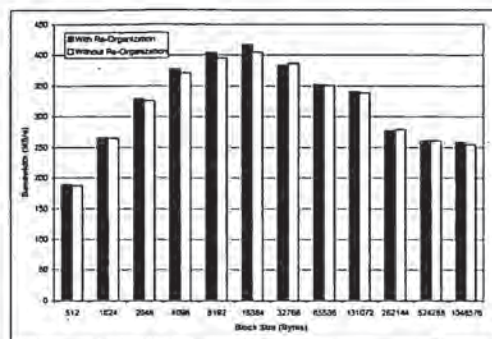


Figure 15. Impact of Buffer Re-Organization

finiBand clusters. The original iSCSI protocol is based on TCP/IP, which means that it can take advantage of existing networking infrastructure and reduce maintenance and management overhead. However, it also suffers from performance inefficiencies in the TCP/IP protocol such as extra copies. Our design offloads data transfer in iSCSI to InfiniBand RDMA operations. Thus, the communication protocol in iSCSI can be simplified and extra buffering in TCP/IP can be avoided, resulting in higher performance and less host overhead. This study can shed light on what can be expected from iSER, the protocol newly proposed to take advantage of RDMA operations.

We have discussed the basic idea and detailed design and implementation issues related to RDMA assisted iSCSI. A prototype has been implemented for the Linux operating system. Our performance evaluation shows that with RDMA assisted iSCSI, we can improve file read bandwidth from around 96 MB/s to over 417 MB/s. Our results also show that RDMA assisted iSCSI can reduce storage latency and host overhead. We have also found that handling buffer registration cost is very important in RDMA based designs. Unless this issue is addressed properly, we cannot obtain any benefit from RDMA.

Currently, we have implemented a RAM disk based target for our RDMA assisted iSCSI. In future, we plan to enhance this design by allowing access to real disks. We are also planning to carry out more performance evaluation by using real world applications. Another direction we are currently working on is to study the benefit of offloading iSCSI control packets from TCP/IP to InfiniBand.

Acknowledgments

We would like to thank Bulent Abali of IBM Research for valuable discussions and his support.

References

- [1] IBAL: InfiniBand Linux SourceForge Project. <http://infiniband.sourceforge.net/IAL/Access/IBAL>.
- [2] Intel iSCSI SourceForge Project. <http://sourceforge.net/projects/intel-iscsi>.
- [3] IP over InfiniBand Working Group. <http://www.ietf.org/html.charters/ipoib-charter.html>.
- [4] M. Chadalapaka, H. Shah, U. Elzur, P. Thaler, and M. Ko. A study of iSCSI extensions for RDMA (iSER). In *ACM SIGCOMM workshop on Network-I/O convergence: experience, lessons, implications*, August 2003.
- [5] J. Chase, A. Gallatin, and K. Yocum. End system optimizations for high-speed TCP. *IEEE Communications Magazine*, 39(4):68–74, 2001.
- [6] DAFS Collaborative. Direct Access File System Protocol, V1.0, August 2001.
- [7] DAT Collaborative. uDAPL and kDAPL API Specification V1.0, June 2002.
- [8] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle. The Direct Access File System. In *2nd USENIX Conference on File and Storage Technologies (FAST '03)*, March 2003.
- [9] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, pages 66–76, March/April 1998.
- [10] D. Gilbert. The Linux SCSI Subsystem in 2.4. http://www.torque.net/scsi/linux.scsi_24/.
- [11] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 24 2000.
- [12] J. Liu, B. Chandrasekaran, J. W. W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. K. Panda. Performance Comparison of MPI Implementations over InfiniBand Myrinet and Quadrics. In *Supercomputing 2003: The International Conference for High Performance Computing and Communications*, Nov. 2003.
- [13] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and performance of the direct access file system. In *Proceedings of USENIX 2002 Annual Technical Conference, Monterey, CA*, pages 1–14, June 2002.
- [14] Mellanox Technologies. Mellanox InfiniBand InfiniHost Adapters, July 2002.
- [15] K. Z. Meth and J. Satran. Design of the iSCSI Protocol. In *20th IEEE Symposium on Mass Storage Systems*, 2003.
- [16] J. C. Mogul. TCP Offload Is a Dumb Idea whose Time Has Come. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, May 2003.
- [17] Oracle. Achieving Main-Frame Class Performance on Intel Servers Using InfiniBand Building Blocks. <http://otn.oracle.com/deploy/availability/pdf/oracle.IB.pdf>.
- [18] A. Palekar, N. Ganapathy, A. Chadda, and R. D. Russell. Design and implementation of a Linux SCSI target for storage area networks. In *5th Annual Linux Showcase and Conference*, November 2001.
- [19] RDMA Consortium. iSCSI Extensions for RDMA (iSER) and Datamover Architecture for iSCSI (DA) Specifications, 2003.
- [20] RDMA Consortium. iWARP Protocol Suite Specifications, 2003.
- [21] P. Sarkar, S. Uttamchandani, and K. Voruganti. Storage Over IP: When Does Hardware Support Help? In *2nd USENIX Conference on File and Storage Technologies (FAST '03)*, March 2003.
- [22] P. Sarkar and K. Voruganti. IP Storage: The Challenges Ahead. In *19th IEEE Symposium on Mass Storage Systems*, 2002.
- [23] Technical Committee T10. SCSI RDMA Protocol, 2002.
- [24] The Internet Engineering Task Force. IP Security Protocol. <http://www.ietf.org/html.charters/ipsec-charter.html>.
- [25] The Internet Engineering Task Force. IP Storage Protocols. <http://www.ietf.org/html.charters/ips-charter.html>, 2002.
- [26] The Internet Engineering Task Force. iSCSI Specification, 2002.
- [27] Voltaire Inc. High Performance SAN Connectivity for InfiniBand Fabrics. http://www.voltaire.com/pdf/-storage.wp_final.pdf.

On the Provision of Prioritization and Soft QoS in Dynamically Reconfigurable Shared Data-Centers over InfiniBand*

P. Balaji S. Narravula K. Vaidyanathan H. -W. Jin D. K. Panda

Computer Science and Engineering,
The Ohio State University,
2015 Neil Avenue,
Columbus, OH 43210

{balaji, narravul, vaidyana, jinhy, panda}@cse.ohio-state.edu

Abstract

In the past few years several researchers have proposed and configured data-centers providing multiple independent services, known as shared data-centers. For example, several ISPs and other web service providers host multiple unrelated web-sites on their data-centers allowing potential differentiation in the service provided to each of them. Such differentiation becomes essential in several scenarios in a shared data-center environment. In this paper, we extend our previous work to provide dynamic reconfigurability of the nodes in the data-center environment; we propose extensions to allow differentiation in the service provided to the different websites in the shared data-center. We propose two extensions to the basic dynamic reconfigurability scheme, namely (i) *Dynamic Reconfiguration with Prioritization* and (ii) *Dynamic Reconfiguration with Prioritization and Soft QoS*. Our experimental results show that our extensions can allow the dynamic reconfigurability scheme to attain a performance improvement of up to 5 times for high priority websites irrespective of any background low priority requests. Further, they can also significantly improve the performance of low priority requests when there are minimal or no high priority requests in the system. Finally, they can achieve a similar performance as a static scheme with up to 50% lesser nodes in some cases.

Keywords: Dynamic Reconfigurability, Service Differentiation, Soft QoS, InfiniBand, Clusters, Data-Centers

1 Introduction

Cluster computing systems are becoming increasingly popular for providing *cost-effective* and *affordable* computing environments for a wide range of applications. These systems are typically built by interconnecting a set of com-

modity PCs or workstations using *high-speed interconnects*. During the last few years, research and industry communities have been proposing and implementing several high performance communication systems to address some of the problems associated with the traditional networking protocols for cluster-based systems. InfiniBand Architecture (IBA) [2] has been recently standardized by the industry to design next generation high-end clusters.

IBA is envisioned as the default interconnect for several environments in the near future. IBA relies on two key features, namely *User-level Networking* and *One-Sided Communication Operations*. User-level Networking allows applications to directly and safely access the network interface without going through the operating system. One-sided communication allows the network interface to transfer data between local and remote memory buffers without any interaction with the operating system or processor intervention. It also provides features for performing network based atomic operations on the remote memory regions. These can be leveraged in providing efficient support for multiple environments [18, 28].

On the other hand, with the increasing adoption of Internet as the primary means of interaction and communication, highly scalable and available web servers have become a critical requirement. Based on these two trends, researchers and industries have proposed the feasibility and potential of cluster-based data-centers [25, 11, 5, 20].

Figure 1 represents a typical cluster-based data-center. The various nodes in the traditional data-center are logically partitioned to provide various related services including web and messaging services, transaction processing, business logic, databases, etc. These nodes interact with each other depending on the query to provide the service requested by the end user.

In the past few years several researchers have proposed and configured data-centers providing multiple independent ser-

*This research is supported in part by Department of Energy's Grant #DE-FC02-01ER25506, and National Science Foundation's grants #CCR-0204429, and #CCR-0311542

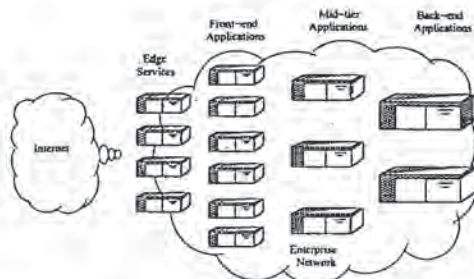


Figure 1. A Typical Cluster-Based Data-Center (courtesy CSP Architecture design [25])

vices, known as shared data-centers [9, 16]. For example, several ISPs and other web service providers host multiple unrelated web-sites on their data-centers allowing potential differentiation in the service provided to each of them. Such differentiation becomes essential in several scenarios in a shared data-center environment. For example, a data-center may want to give a higher priority to all requests pertaining to website A (a high paying customer) as compared to website B (a low paying customer).

Similarly, a data-center might guarantee a certain Quality of Service (QoS) guarantees in the resources provided to each service or website it is hosting. In the context of data-centers, we classify QoS guarantees into two classes: *Hard QoS guarantees* and *Soft QoS guarantees*. Hard QoS guarantees require that the resources guaranteed be available to the website whenever required. On the other hand, Soft QoS guarantees rely on the average load on the website and client workload pattern studies. For example, though traffic to a website could be bursty at times requiring a large number of nodes, the average load on the website could be much lesser requiring fewer nodes. In this paper, we focus on providing *Soft QoS guarantees* among other features for shared data-centers.

Statically assigning the nodes in the shared data-center for each service provided depending on the priority or the soft QoS guarantees given to the website is a widely used approach. In this approach, nodes are allotted to each service depending either on the worst case estimates of the load expected or on the soft QoS guarantees provided together with the nodes available in the data-center. It is easy to see that this approach is easily capable of meeting the priority of soft QoS requirements given by the data-center. However, this approach has several disadvantages. Firstly, it might incur severe under utilization of resources especially when the traffic is bursty and directed to a single web-site. Secondly, this approach restricts the QoS guarantees provided by the data-center based on the number of nodes present. Finally, this approach can result in a poor performance for the low priority requests even when there are no high priority requests available in the system.

ity requests available in the system.

In our previous work [4], we presented a novel design to provide dynamic reconfigurability of the nodes in the data-center environment. This technique enables the nodes in the data-center environment to efficiently adapt their functionality based on the system load and traffic pattern. Dynamic reconfigurability attempts to provide benefits in several directions: (i) cutting down the time needed for configuring and assigning the resources available by dynamically transferring the traffic load to the best available node/server, (ii) improving the performance achievable by the data-center by reassigning under-utilized nodes to loaded services, (iii) cutting down the cost of the data-center by reducing over-provisioning of nodes and improving the utilization of the resources available inside the data-center and several others.

While reconfigurability is a widely used technique for clusters, the data-center environment poses several interesting challenges for the design and implementation of such a scheme. In our approach, we used the advanced features of InfiniBand such as Remote Direct Memory Access (RDMA) operations and network based atomic operations to tackle these challenges in an efficient manner without requiring any modifications to the existing data-center applications.

However, as we will see in Section 4, the basic dynamic reconfigurability scheme does not have any concept of service differentiation *per se* and cannot be *directly* used in a shared data-center environment having different service requirements for different websites.

In this paper, we extend our basic dynamic reconfigurability scheme to allow service differentiation in the shared data-center environment. In particular, we point out the issues associated with the basic dynamic reconfigurability scheme and propose two extensions to it, namely (i) *Dynamic Reconfiguration with Prioritization* and (ii) *Dynamic Reconfiguration with Prioritization and Soft QoS*.

This work has several research contributions. Primarily, we extend on our previous work [5, 20, 4] in understanding the role of the InfiniBand architecture and other RDMA capable networks in next-generation data-centers. The other main contributions are:

1. While our previously proposed dynamic reconfigurability approach allows high performance for shared data-centers by improving the overall resource utilization in the system, it suffers from a lack of service differentiation capability. In this paper, we point out several issues associated with this and other such dynamic reconfigurability schemes.
2. We propose extensions to the dynamic reconfigurability scheme making it service differentiation capable and suitable for practical usability in shared data-center environments. Further, this architecture requires

no changes to the legacy data-center applications. The legacy data-center applications have to be aware of neither the dynamic reconfiguration of nodes between different websites, nor of service differentiation provided by our scheme to be utilized by the shared data-center.

3. As we will see in the next few sections, though a reconfigurability scheme can be implemented based on the standard sockets interface, the high computation load typical in the data-center environment can significantly hurt the performance achieved. With the focus on this observation, we design our approach to utilize the remote memory capabilities of the InfiniBand architecture to remotely monitor and manage the nodes in the data-center environment. Our results also show that this one-sided communication based architecture is mostly resilient and well-conditioned to the load on the servers as compared to two-sided protocols such as TCP/IP (sockets).

We evaluated our proposed schemes on an InfiniBand based cluster and attempted to bring out the benefits and issues associated with these schemes. Our experimental results show that our extensions can allow the dynamic reconfigurability scheme to attain a performance improvement of up to 5 times for high priority websites irrespective of any background low priority requests. Further, our extensions can also significantly improve the performance of low priority requests when there are minimal or no high priority requests in the system. Finally, our extensions can achieve a similar performance as a static scheme with up to 50% lesser nodes in some cases.

The remaining part of the paper is organized as follows: Section 2 provides a brief background of the InfiniBand architecture and shared data-center environments. In Section 3, we describe the details about the design and implementation of our basic dynamic reconfiguration scheme. In Section 4 we describe the provisioning of prioritization and soft QoS guarantees in shared data-centers and our extensions to the dynamic reconfigurability approach to provide prioritization and soft QoS guarantees. We describe our experimental results in Section 5, present some previous related work in Section 6 and draw our conclusions and possible future work in Section 7.

2 Background

In this section, we provide a brief background on (i) the InfiniBand architecture and its advanced one-sided operations like RDMA and atomic operations and (ii) the Clustered data-center environment.

2.1 InfiniBand

The InfiniBand Architecture (IBA) is an industry standard that defines a System Area Network (SAN) to design clus-

ters offering low latency and high bandwidth. The compute nodes are connected to the IBA fabric by means of Host Channel Adapters (HCAs). IBA defines a semantic interface called as Verbs for the consumer applications to communicate with the HCAs. VAPI is one such interface developed by Mellanox Technologies.

2.1.1 RDMA Communication Model

IBA supports two types of communication semantics: channel semantics (send-receive communication model) and memory semantics (RDMA communication model).

In channel semantics, every send request has a corresponding receive request at the remote end. Thus, there is one-to-one correspondence between every send and receive operation.

In memory semantics, RDMA operations are used. These operations are transparent at the remote end, i.e., they do not require the remote end to be involved in the communication. Therefore, an RDMA operation has to specify both the memory address for the local buffer as well as that for the remote buffer. There are two kinds of RDMA operations: RDMA Write and RDMA Read. In an RDMA write operation, the initiator directly writes data into the remote node's user buffer. Similarly, in an RDMA Read operation, the initiator reads data from the remote node's user buffer.

2.1.2 Atomic Operations Over IBA

In addition to RDMA, the reliable communication classes also optionally include atomic operations directly against the memory at the end node. Atomic operations are posted as descriptors as in any other type of communication. However, the operation is completely handled by the HCA. The atomic operations supported are Fetch-and-Add and Compare-and-Swap, both on 64-bit data. The Fetch-and-Add operation performs an atomic addition at the remote end. The Compare-and-Swap is used to compare two 64-bit values and swap the remote value with the data provided if the comparison succeeds.

2.2 Shared Data-Center Environment

A clustered data-center environment essentially tries to utilize the benefits of a cluster environment (e.g., high performance-to-cost ratio) to provide the services requested in a data-center environment (e.g., web hosting, transaction processing). As mentioned earlier, researchers have proposed and configured data-centers to provide multiple independent services, such as hosting multiple web-sites, forming what is known as shared data-centers.

Figure 2 shows a higher level layout of a shared data-center architecture hosting multiple web-sites. External clients request documents or services from the data-center over the WAN/Internet through load-balancers using higher

level protocols such as *HTTP*. The load-balancers on the other hand serve the purpose of exposing a single IP address to all the clients while maintaining a list of several internal IP addresses to which they forward the incoming requests based on a pre-defined algorithm (e.g., round-robin).

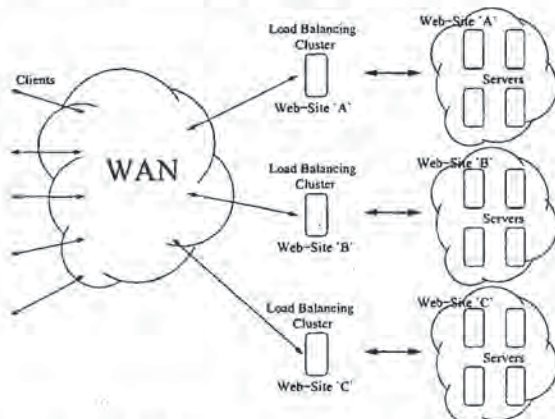


Figure 2. A Shared Cluster-Based Data-Center Environment

While hardware load-balancers are commonly available today, they suffer from being based on a pre-defined algorithm and are difficult to be tuned based on the requirements of the data-center. On the other hand, though software load-balancers are easy to modify and tuned based on the data-center requirements, they can potentially form bottlenecks themselves for highly loaded data-centers. In the past, several researchers have proposed the use of an additional cluster of nodes (known as the edge tier) [25] to perform certain services such as intelligent load-balancing, caching, etc [16]. Requests can be forwarded to this cluster of software load-balancers either by the clients themselves by using techniques such as DNS aliasing, or by using an additional hardware load-balancer.

The servers inside the clustered data-center provide the actual services such as web-hosting, transaction processing, etc. Several of these services require computationally intensive processing such as CGI scripts, Java servlets and database query operations (table joins, etc). This makes the processing on the server nodes CPU intensive in nature.

3 Dynamic Reconfiguration Overview

In our previous work [4], we presented a novel design to provide dynamic reconfigurability of the nodes in the data-center environment. This technique enables the nodes in the data-center environment to efficiently adapt their functionality based on the system load and traffic pattern. However, as we will see in Section 4, the basic dynamic reconfigurability

scheme does not have the concept of service differentiation *per se* and cannot be *directly* used in a shared data-center environment having different service requirements for different websites.

In this section, we briefly describe the basic design issues in the dynamic reconfigurability scheme and the details about the implementation of this scheme using the native Verbs layer over InfiniBand (VAPI). In Section 4, we propose several extensions to the basic dynamic reconfigurability scheme to allow service differentiation in the shared data-center environment.

3.1 Reconfigurability Support

Request patterns seen over a period of time, by a shared data-center, may vary significantly in terms of the ratio of requests for each co-hosted web-site. For example, interesting documents or dynamic web-pages becoming available and unavailable might trigger bursty traffic for some web-site at some time and for some other web-site at a different time. This naturally changes the resource requirements of a particular co-hosted web site from time to time. The basic idea of reconfigurability is to utilize the idle nodes of the system to satisfy the dynamically varying resource requirements of each of the individual co-hosted web-sites in the shared data-center. Dynamic reconfigurability of the system requires some extent of functional equivalence between the nodes of the data-center. We provide this equivalence by enabling software homogeneity such that each node is capable of belonging to any web-site in the shared data-center. Depending on the current demands (e.g., due to a burst of requests to one web-site), nodes reconfigure themselves to support these demands.

Support for Existing Applications: A number of applications have been developed in the data-center environment over the span of several years to process requests and provide services to the end user; modifying them to allow dynamic reconfigurability is impractical. To avoid making these cumbersome changes to the existing applications, our design makes use of *external helper modules* which work alongside the applications to provide effective dynamic reconfiguration. Tasks related to system load monitoring, maintaining global state information, reconfiguration, etc. are handled by these helper modules in an application transparent manner. These modules, running on each node in the shared data-center, reconfigure nodes in the data-center depending on current request and load patterns. They use the run-time configuration files of the data-center applications to reflect these changes. The servers on the other hand, just continue with the request processing, unmindful of the changes made by the modules.

Load-Balancer Based Reconfiguration: Two different approaches could be taken for reconfiguring the nodes: Server-based reconfiguration and Load-balancer based re-

configuration. In server-based reconfiguration, when a particular server detects a significant load on itself, it tries to reconfigure a relatively free node that is currently serving some other web-site content. Though intuitively the loaded server itself is the best node to perform the reconfiguration (based on its closeness to the required data and the number of messages required), performing reconfiguration on this node adds a significant amount of load to an already loaded server. Due to this reason, reconfiguration does not happen in a timely manner and the overall performance is affected adversely. On the other hand, in load-balancer based reconfiguration, the edge servers (functioning as load-balancers) detect the load on the servers, find a free server to alleviate the load on the loaded server and perform the reconfiguration themselves. Since the shared information like load, server state, etc. is closer to the servers, this approach incurs the cost of requiring more network transactions for its operations.

Remote Memory Operations Based Design: As mentioned earlier, by their very nature the server nodes are compute intensive. Execution of CGI-Scripts, business-logic, servlets, database processing, etc. are typically very taxing on the server CPUs. So, the helper modules can potentially be starved for CPU on these servers. Though in theory the helper modules on the servers can be used to share the load information through explicit two-sided communication, in practice, such communication does not perform well [20].

InfiniBand, on the other hand, provides one-sided remote memory operations (like RDMA and Remote Atomics) that allow access to remote memory without interrupting the remote node. In our design, we use these operations to perform load-balancer based server reconfiguration in a server transparent manner. Since the load-balancer is performing the reconfiguration with no interruptions to the server CPUs, this RDMA based design is highly resilient to server load.

The major design challenges and issues involved in dynamic adaptability and reconfigurability of the system are listed below.

- Providing a System Wide Shared State
- Concurrency Control to avoid Live-locks and Starvation
- Avoiding server thrashing through history aware reconfiguration
- Tuning the reconfigurability module sensitivity

Figure 3 shows the RDMA based protocol used by Dynamic Reconfigurability. As shown in the figure, the entire cluster management and dynamic reconfiguration is performed by the lightly loaded load-balancer nodes without disturbing the server nodes using the RDMA and remote

atomic operations provided by InfiniBand. Further details about the other design issues can be found in [4].

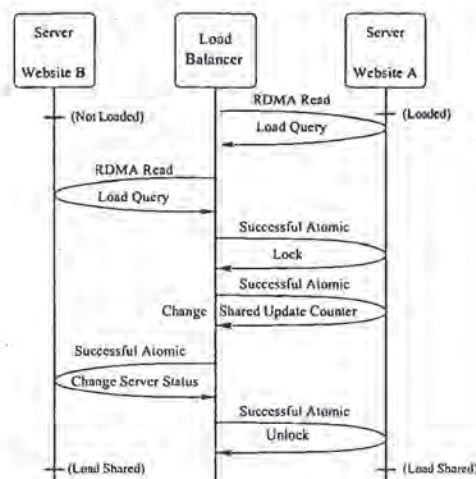


Figure 3. RDMA based Protocol for Dynamic Reconfigurability

4 Service Differentiation with Dynamic Reconfigurability

In this section, we point out the issues associated with the dynamic reconfigurability scheme and propose extensions to this scheme to allow service differentiation in a shared data-center environment.

4.1 Service Differentiation

As mentioned earlier, differentiation in service becomes essential in several scenarios in a shared data-center environment. For example, a data-center may want to give a higher priority to all requests pertaining to website A (a high paying customer) as compared to website B (a low paying customer). Similarly, a data-center might guarantee a certain Quality of Service (QoS) guarantees in the resources provided to each service or website it is hosting.

Statically assigning the nodes in the shared data-center for each service provided depending on the priority or the soft QoS guarantees given to the website is a widely used approach. In this approach, nodes are allotted to each service depending either on the worst case estimates of the load expected or on the soft QoS guarantees provided together with the nodes available in the data-center. It is easy to see that this approach is easily capable of meeting the priority of soft QoS requirements given by the data-center. However, this approach has several disadvantages. Firstly, it might incur severe under utilization of resources especially when

the traffic is bursty and directed to a single web-site. Secondly, this approach restricts the QoS guarantees provided by the data-center based on the number of nodes present. Finally, this approach can result in a poor performance for the low priority requests even when there are no high priority requests available in the system.

Issues with Dynamic Reconfigurability: In the dynamic reconfigurability approach the nodes in the data-center environment adapt their functionality based on the system load and traffic pattern. This allows a higher utilization of the system resources and essentially improves the overall performance of the system. However, this scheme does not have concept of service differentiation *per se* and cannot be directly used in a shared data-center environment having different service requirements for different websites. The scheme performs reconfiguration only based on the load on the physical nodes and does not consider any prioritization and QoS guarantees the website might have to meet.

This means that a burst of requests to one website would result in all the nodes in the data-center to be re-assigned to this website. Now, while this website is still loaded, if there is a burst of requests to another higher priority website, the reconfigurability scheme will not be able to find a lightly loaded physical node and thus will not perform any additional reconfigurations to handle the new burst of high priority requests.

Figure 4 depicts this issue with the basic dynamic reconfigurability mechanism in a shared data-center environment hosting two websites where one of the websites is a high priority website and the other is a lower priority website.

Figure 4a shows the initial state of the system where both the websites have a small amount of requests coming in. Both websites are initially configured with four nodes each. Figure 4b shows a state where there is a heavy load of requests to website B. Since website A is not that heavily loaded, the nodes servicing this website are reconfigured to serve website B. Figure 4c shows a state where there is a heavy load of requests for website A, while website B is still heavily loaded. In this scenario, since website A cannot find any lightly loaded servers, it does not perform any reconfiguration of nodes, causing a degradation of performance for the high priority website.

However, ideally we would like this burst of high priority requests to reconfigure the nodes from the lower priority website to handle the requests on the high priority website.

4.2 Schemes to provide Service Differentiation

Statically assigning resources based on the service requested by the website, is a commonly used approach for providing service differentiation in shared data-centers. We briefly describe two schemes to provide such service differentiation, namely (i) *Rigid* and (ii) *Dynamic Reconfiguration based Service Differentiation*.

4.2.1 Rigid Scheme

In the rigid scheme, the data-center statically assigns the available nodes in the data-center as requested by the websites. For example, if the data-center has eight nodes and hosts two websites (website A and website B), the number of nodes it can commit to both the websites together cannot exceed eight nodes. For example, six nodes to website A and two nodes to website B is one possible node guarantee the data-center can give in this model.

We use two instances of this scheme in our evaluation: *Rigid-Small* and *Rigid-Large*. Both instances are exactly the same except that *Rigid-Large* uses a larger number of physical nodes as compared to *Rigid-Small*.

4.2.2 Dynamic Reconfiguration based Service Differentiation

We propose two extensions to the basic dynamic reconfigurability scheme, namely (i) *Dynamic Reconfiguration with Prioritization* and (ii) *Dynamic Reconfiguration with Prioritization and Soft QoS* to allow dynamic reconfigurability to do re-allocation of nodes while considering the service differentiation requirements provided by the data-center.

Dynamic Reconfiguration with Prioritization: In the prioritization scheme, the reconfigurability modules keep track of the priority-level to which each website belongs. On detecting a high load, the scheme allows reconfiguring another node if either it is lightly loaded or it belongs to a lower priority level. Figure 5 depicts the previous example with this scheme.

Figure 5a shows the initial state of the system where both the websites have a small amount of requests coming in. Again, both websites are initially configured with four nodes each. Figure 5b shows a state where there is a heavy load of requests to website B. Similar to the basic reconfigurability scheme, since website A is not that heavily loaded at this time, the nodes servicing this website are reconfigured to serve website B. Figure 5c shows a state where there is a heavy load of requests for website A, while website B is still heavily loaded. Unlike the basic reconfigurability scheme, in this scenario since website B is a low priority website, with the dynamic reconfiguration with prioritization scheme the nodes can be reconfigured to serve website A. So, as the figure depicts, the nodes are reconfigured to serve website A thus improving the performance of the high priority website. Now however, website B only has one node left to serve its pertinent incoming requests; this could lead to starvation for requests for website B. In summary, though the dynamic reconfigurability with prioritization scheme can give a higher performance to the high priority websites as compared to the basic reconfigurability scheme, it might result in starvation for the low priority requests.

Dynamic Reconfiguration with Prioritization and Soft

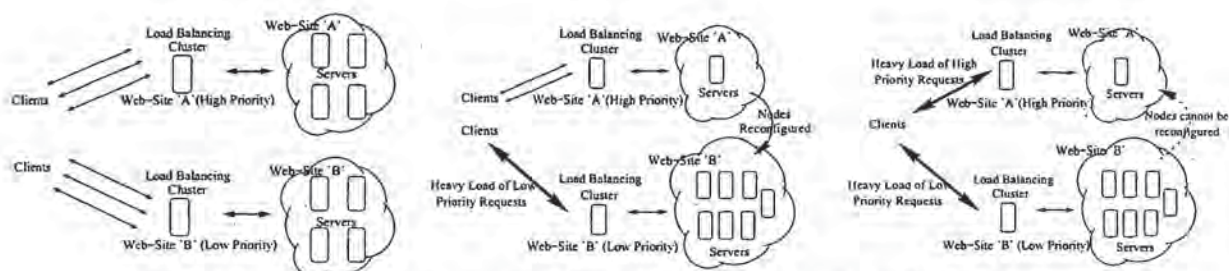


Figure 4. Basic Dynamic Reconfigurability: (a) Step 1, (b) Step 2 and (c) Step 3

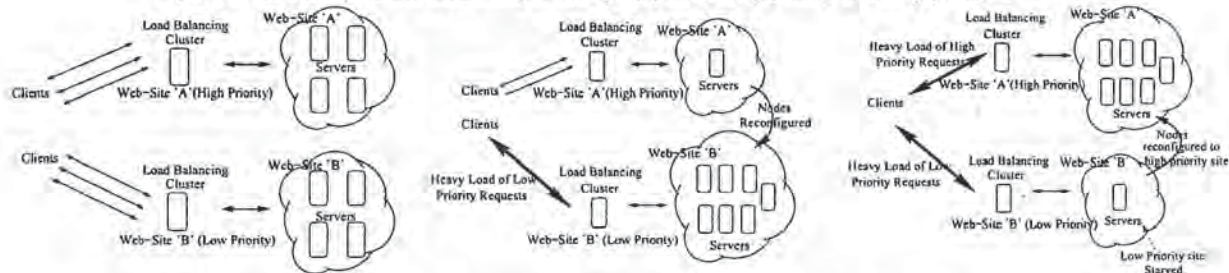


Figure 5. Dynamic Reconfigurability with Prioritization: (a) Step 1, (b) Step 2 and (c) Step 3

QoS: The dynamic reconfiguration with prioritization scheme allows the higher priority requests to reconfigure nodes from the lower priority sites and thus increase the performance of these requests. However, just shifting the nodes based on priorities might cause lower priority requests to be starved. For example, in Figure 5c, the lower priority requests only have one node to send requests to.

In the dynamic reconfiguration with prioritization and soft QoS scheme, each website has fixed QoS guarantees. This scheme converts nodes from lower priority websites only if the QoS guarantee of the higher priority website is not met. If the QoS guarantee of the higher priority website is already met, no further reconfiguration is done. Figure 6 depicts the behavior of this scheme in the previous example.

Figure 6a shows the initial state of the system where both the websites have a small amount of requests coming in. Again, both websites are initially configured with four nodes each; each website is provided a soft QoS guarantee of six nodes. Figure 6b shows a state where there is a heavy load of requests to website B. Similar to the previous schemes, since website A is not that heavily loaded at this time, the nodes servicing this website are reconfigured to serve website B. Figure 6c shows a state where there is a heavy load of requests for website A, while website B is still heavily loaded. This scheme is similar to the previous scheme in the sense that when the high priority requests arrive, the scheme reconfigures nodes serving the lower priority website B to serve the higher priority website A. However, the scheme reconfigures nodes only till the QoS

guarantee of website A is met, i.e., it reconfigures nodes till website A has six nodes serving it. This ensures that the lower priority website (website B) is ensured of having at least two nodes (in this example). It is to be noted that, by changing the QoS guarantees of the website A, the amount of starvation of website B can be controlled in this scheme. For example, if website A was only provided with a QoS guarantee of five nodes, website B could utilize three nodes in such a scenario.

5 Experimental Results

In this section, we present various performance results. First, in Section 5.1, we present the ideal case raw performance achievable by the native Verbs API (VAPI) over InfiniBand and TCP/IP over InfiniBand (IPoIB) using microbenchmark results. In Section 5.2, we present the impact of the load conditions in the data-center environment on the performance achievable by VAPI and IPoIB. Finally, in Section 5.3 we present the evaluation of our VAPI based reconfiguration scheme in a shared data-center environment.

For all our experiments we used two clusters whose descriptions are as follows:

Cluster1: A cluster system consisting of 8 nodes built around SuperMicro SUPER P4DL6 motherboards and GC chipsets which include 64-bit 133 MHz PCI-X interfaces. Each node has two Intel Xeon 2.4 GHz processors with a 512 kB L2 cache and a 400 MHz front side bus and 512 MB of main memory. We used the RedHat 9.0 Linux distribution and Linux-2.4.22smp kernel.org kernel.

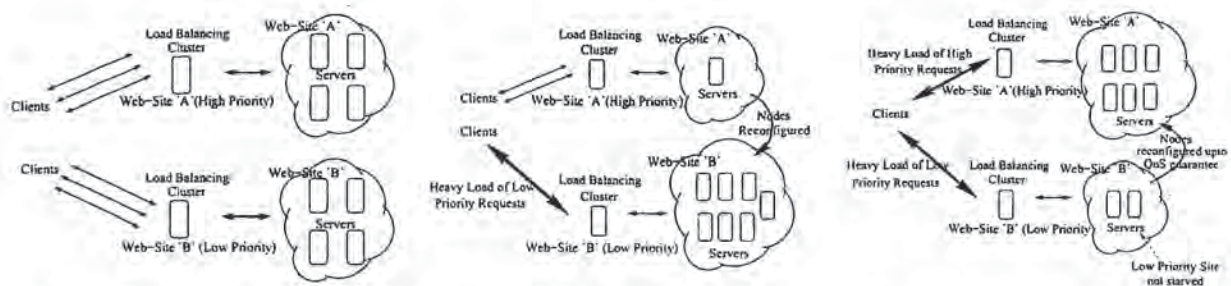


Figure 6. Dynamic Reconfigurability with Prioritization and Soft QoS guarantees: (a) Step 1, (b) Step 2 and (c) Step 3

Cluster2: A cluster system consisting of 8 nodes built around SuperMicro SUPER X5DL8-GG motherboards with ServerWorks GC LE chipsets which include 64-bit 133 MHz PCI-X interfaces. Each node has two Intel Xeon 3.0 GHz processors with a 512 kB L2 cache and a 533 MHz front side bus and 512 MB of main memory. We used the RedHat 9.0 Linux distribution and Linux-2.4.22smp kernel.org kernel.

The following interconnect was used to connect all the nodes in Clusters 1 and 2.

Interconnect: InfiniBand network with Mellanox InfiniHost MT23108 DualPort 4x HCA adapter through an InfiniScale MT43132 twenty-four 4x Port completely non-blocking InfiniBand Switch. The Mellanox InfiniHost HCA SDK version is thca-x86-3.2. The adapter firmware version is fw-23108-rel-3.2.0-rc4-build-001. The IPoIB driver for the InfiniBand adapters was provided by Mellanox Corporation. The driver was obtained from Mellanox Golden CD version 0.5.0.

Cluster 2 was used to represent the software load-balancers and Cluster 1 was used to represent the server nodes in the data-center environment. We used Apache version 2.0.50 in all our data-center experiments. For dynamic content requests, the application server was configured using PHP version 4.3.7 and the database used was MySQL version 4.1 with Master-Slave clustering for INNODB table type. Requests from the software load-balancers were generated using sixteen threads on each load-balancer.

5.1 Basic Microbenchmarks

The ideal case performance achievable by the Verbs API (VAPI) over InfiniBand and IPoIB using micro-benchmark tests are presented in this section. VAPI provides multiple communication models for transferring data namely: (a) Send-Receive, (b) RDMA write, (c) RDMA write with immediate data and (d) RDMA Read. In this paper, we show the performance achieved by the RDMA Read communication model and that achieved by TCP/IP over InfiniBand

(IPoIB). Results for the other communication models can be found in [3].

InfiniBand provides two mechanisms for completion notification. The first approach is polling-based which requires the host application to continuously poll on the completion queue and check for the completion of the message transmission or reception. The second approach is notification-based which allows the host application to request an interrupt based notification from the network adapter. The notification based approach incurs the additional cost of an interrupt. The polling based approach does not incur this additional cost, but results in a high CPU utilization due to the continuous polling of the completion queue. In this section, we present results for both the polling based approach as well as the notification based approach on Cluster 1.

The latency achieved by the VAPI RDMA Read communication model and IPoIB (round-trip latency) for various message sizes is shown in Figure 7a. RDMA Read, using the polling based approach, achieves a latency of $11.89\mu s$ for 1 byte messages compared to the round trip latency of $53.8\mu s$ achieved by IPoIB. The event based approach, however, achieves a latency of $23.97\mu s$. Further, with increasing message sizes, the difference between the latency achieved by VAPI and IPoIB tends to increase. The figure also shows the CPU utilized by RDMA Read (notification based) and IPoIB. We can see that RDMA utilizes negligible CPU on the receiver side, i.e., with RDMA, the initiator can read or write data from the remote node without requiring any interaction with the remote host CPU.

Figure 7b shows the uni-directional bandwidth achieved by RDMA Read and IPoIB. RDMA Read (with polling as well as with notification) is able to achieve a peak bandwidth of 839.1 MBps as compared to a 231 MBps achieved by IPoIB. Again, the CPU utilization for RDMA is negligible on the receiver side.

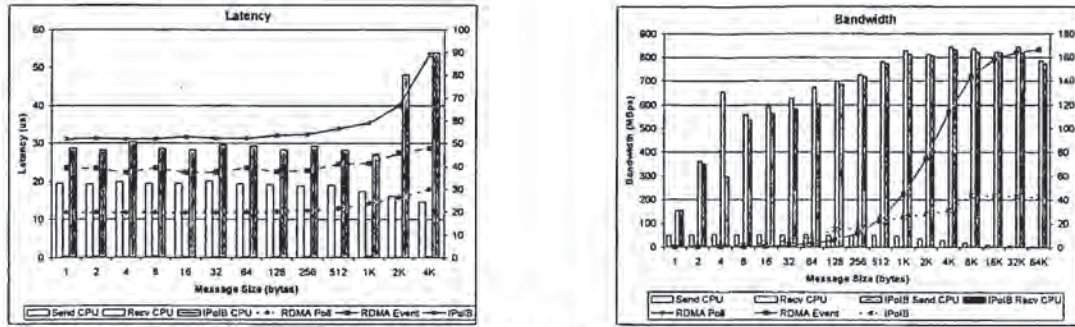


Figure 7. Micro-Benchmarks for RDMA Read and IPoIB: (a) Latency and (b) Bandwidth

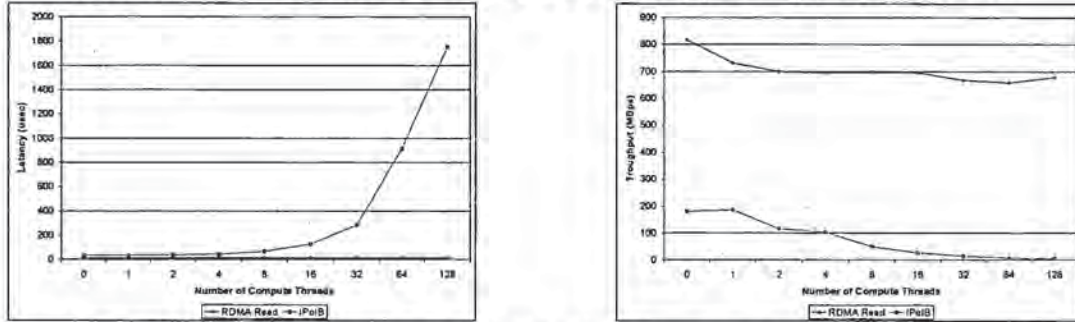


Figure 8. Performance of IPoIB and RDMA Read with background threads: (a) Latency and (b) Bandwidth

5.2 Impact of Background Threads

In this section, we present performance results showing the impact of the loaded conditions in the data-center environment on the performance of RDMA Read and IPoIB on Cluster 1. The results for the other communication models can be found in [3].

We emulate the loaded conditions in the data-center environment by performing background computation and communication operations on the server while the load-balancer performs the test with a separate thread on the server. This environment emulates a typical cluster-based shared data-center environment where multiple server nodes communicate periodically and exchange messages, while the load balancer, which is not as heavily loaded, attempts to get the load information from the heavily loaded machines.

The performance comparison of RDMA Read and IPoIB for this experiment is shown in Figure 8. We observe that the performance of IPoIB degrades significantly with the increase in the background load. On the other hand, one-sided communication operations such as RDMA show absolutely no degradation in the performance. These results show the capability of one-sided communication primitives in the data-center environment.

5.3 Evaluation of Dynamic Reconfigurability

In this section, we evaluate the basic Dynamic Reconfigurability scheme as well as our extensions to the scheme, namely *Dynamic Reconfigurability with Prioritization* and *Dynamic Reconfigurability with Prioritization and Soft QoS* in several different scenarios. We first show the basic performance achieved by the dynamic reconfigurability scheme in Section 5.3.1. In Section 5.3.2, we evaluate the capabilities of the basic dynamic reconfigurability scheme in a shared data-center hosting multiple websites of different priority levels and compare it with the performance of the two extensions we proposed in this paper.

For our evaluations, we used three different traces for the basic results. For detailed analysis we used a single file trace with a request file which is 1KB, 4KB or 16KB in size in order to understand the impact of various schemes without being diluted by other system parameters. We also show the applicability of our scheme in a real environment using the ZipF workload benchmark trace [29] and a real-world WorldCup trace [1]. In this paper, we show the evaluation results with a single file trace of 1KB file size and some selected results with the ZipF and WorldCup traces. The rest of the results are skipped due to space constraints and can be found in [3].

5.3.1 Basic Dynamic Reconfigurability

In this section, we present the basic performance achieved by the dynamic reconfigurability scheme.

In this experiment, we consider a shared data-center hosting two websites with equal priorities. We consider two instances of rigid or static node allocation: *Rigid-Small* and *Rigid-Large*. *Rigid-Small* considers a data-center with eight nodes and allots four nodes to each website (both websites are of equal priority). *Rigid-Large* considers a data-center with twelve nodes and allots six nodes to each website. We compare these schemes with the basic dynamic reconfigurability and the Soft QoS based dynamic reconfigurability schemes; these schemes consider a data-center with only eight nodes and dynamically moves the nodes between these websites depending on the load on each website.

Figure 9 shows the performance achieved by these schemes. The x-axis depicts the length of requests to each website. For example, a burst length of 2048 means that the data-center receives a burst of 2048 requests for the first website followed by a burst of 2048 requests for the second website and so on in a round-robin manner. We see that for small burst lengths the dynamic reconfigurability schemes perform comparably with the *Rigid-Small* scheme. As the burst length increases, their performance increases and converges with that of the *Rigid-Large* scheme. This is because, for small burst lengths the dynamic reconfigurability schemes do not have enough time to shift the nodes from the lightly loaded website to the heavily loaded website. As the burst length increases, the schemes have enough time to move the nodes according to the load on the websites. Further, for burst lengths of close to 16K requests, the time taken to shift the nodes is negligible compared to the time for which the requests arrive. This reflects in a better performance for the dynamic reconfigurability schemes for large burst lengths.

For very small burst lengths, the basic dynamic reconfigurability approach (legend Reconf) has a slight drop in performance. This is because of the additional reconfigurations performed by this scheme. In the Soft QoS based dynamic reconfigurability approach (legend Reconf-PQ), these shifts are limited to some extent by the QoS guarantees (since there are no more reconfigurations after the QoS guarantees are met). This causes the Soft QoS based dynamic reconfigurability to give a closer performance to the Rigid approach as compared to the basic reconfigurability scheme. For large burst lengths, the basic dynamic reconfigurability approach can utilize up to seven nodes in the data-center; on the other hand, the soft QoS based dynamic reconfigurability approach utilizes only up to the QoS guarantee number of nodes (six nodes in our case). This causes the basic dynamic reconfigurability scheme to give a slightly higher performance than the soft QoS based reconfigurability for large burst lengths.

To further understand the behavior of dynamic reconfigurability with burst length, we show the node utilization in the data-center for burst lengths of 1K and 16K in Figure 10. As seen in the figure, for a small burst length of 1K requests, the number of nodes used fluctuates rapidly and stays close to four nodes which is the initial number of nodes provided for the dynamic reconfigurability scheme as well as the *Rigid-Small* scheme. However, for a large burst length of 16K, the node utilization is high and close to the maximum¹.

Node utilization numbers for the soft QoS based dynamic reconfigurability scheme are skipped due to space constraints and can be found in [3].

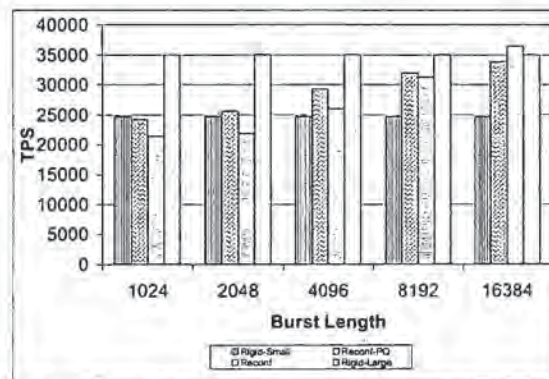


Figure 9. Basic Performance per Web-Site for *Rigid-Small*, Basic Reconfigurability, Reconfigurability with soft QoS and *Rigid-Large*

5.3.2 Service Differentiation with Dynamic Reconfiguration

In this section, we compare the performance attainable by the *Dynamic Reconfigurability with Prioritization* and *Dynamic Reconfigurability with Prioritization and Soft QoS* schemes and compare it with the basic *Dynamic Reconfigurability* scheme. We carry out these evaluations in a shared data-center hosting two websites where one of the websites is a higher priority website compared to the other; each website has a soft QoS guarantee of six nodes. We compare the performance attainable for the high priority requests in terms of the number of transactions the data-center can support as well as the percentage of times it is able to meet the provided QoS guarantee. Further, we compare the starvation extent of the low priority requests using the number of nodes it has available during different scenarios.

¹Seven nodes is the maximum number of nodes a website can obtain since in the dynamic reconfigurability scheme we force at least one node to be serving each website at any point of time

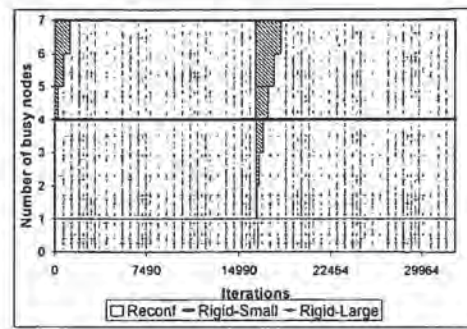
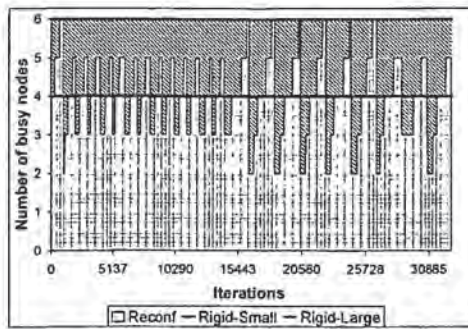


Figure 10. Node Utilization for Reconf (a) Burst Length = 1K and (b) Burst Length = 16K

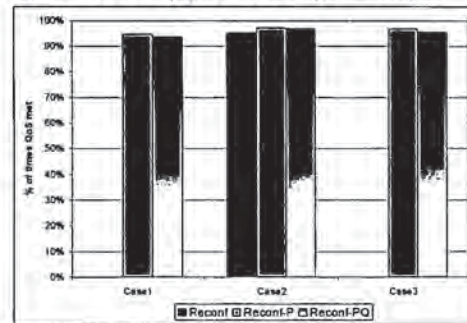
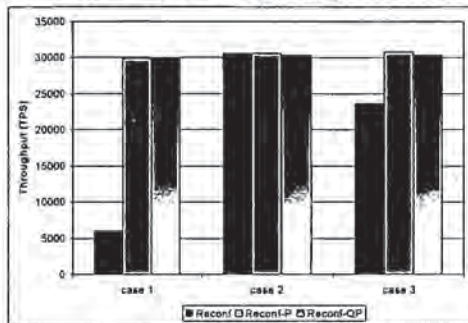


Figure 11. (a) High Priority Web-Site Performance and (b) QoS delivered

In order to evaluate different aspects of the three schemes mentioned, we created three test case scenarios. Though several other scenarios are possible in a production data-center, we believe that these three scenarios would capture the bulk of the characteristic differences between the three schemes.

1. *Case 1:* In this case, we study the behavior of a burst of high priority requests when a background load of low priority requests exists.
2. *Case 2:* In this case, we study the behavior of a load of high priority requests when a burst of low priority requests arrive while the high priority load is running.
3. *Case 3:* In this case, we study the behavior when both high priority requests and low priority requests arrive simultaneously.

Figures 11a and 11b show the basic performance and the QoS meeting capabilities the three schemes achieve for these cases. The analysis of these results for the different cases are as follows:

Case1: As mentioned above, in this case, the data-center first experiences a burst of requests for the low-priority website. While, this burst of low-priority requests is still on, we study the performance of a new burst of requests for the

high-priority website. Figures 12, 13 and 14 show the variation in the number of nodes allocated to the low-priority website for the basic reconfigurability, prioritization and prioritization with soft QoS schemes respectively.

Initially, when there is a burst of low-priority requests, all three schemes allocate more nodes to serve these low priority requests since the load on the high priority server nodes is initially low. However, the basic reconfigurability scheme and the prioritization scheme allocate the maximum number of nodes possible to the low-priority website, while the prioritization with soft QoS scheme allocates nodes up to the QoS guaranteed to that website. In our experiment, the prioritization with soft QoS scheme allocates up to six nodes to each website, while the other two schemes can allocate up to seven nodes.

Now, when there is a burst of high priority requests, each scheme takes a different path. The basic reconfigurability scheme does not have any concept of differentiated service and cannot find any lightly loaded server in the system; thus it does not do any additional reconfiguration. This would result in a degradation of performance of the high priority requests. The prioritization and prioritization with soft QoS schemes, however, allow service differentiation and would reconfigure the nodes from the lower priority website to the higher priority website. Again, the prioritization scheme allocates the maximum number of nodes possible to the high

priority website; the prioritization with soft QoS scheme on the other hand allocates nodes only till the QoS requirement of the website is met, i.e., it allocates a maximum of only six nodes in our experiment. Based on this, the prioritization scheme can be expected to give a slightly better performance as compared to the prioritization with soft QoS scheme for high priority requests. We see that this scheme can achieve up to a factor of five improvement in performance as compared to the basic reconfigurability scheme.

Case2: In this case, the data-center first experiences a burst of requests for the high-priority website. While these requests are on, we study the impact of a burst of low-priority requests on the performance of the high-priority requests. On a burst of high-priority requests, all three schemes initially allocate more nodes to service the high priority website (prioritization with soft QoS only allocates up to the QoS guarantee, while the other schemes allocate up to the maximum possible number of nodes).

Now, when a burst of low-priority requests arrive, the basic reconfigurability scheme does not perform any additional reconfigurations; thus the performance of the high priority requests will not be affected. Similarly, for the prioritization and prioritization with soft QoS schemes, since the incoming requests are of a lower priority, there is no additional re-allocation of nodes. Thus, we expect all three schemes to perform in a similar manner in this case. However, it is to be noted that for the prioritization scheme, the incoming low-priority requests have only one node remaining to service them, while the prioritization with soft QoS scheme has two nodes to service the low priority requests.

Case 3: In this case, the data-center experiences a burst of both high priority and low priority requests at the same time. For the basic dynamic reconfigurability scheme, since all the nodes in the data-center are heavily loaded, there is no re-allocation of nodes. For the prioritization scheme, the maximum possible nodes are re-allocated to the high-priority website. Similarly, for the prioritization with soft QoS scheme, nodes are re-allocated to the high-priority website till the QoS guarantee for the high priority website is met.

The results for the node allocation time line are skipped for Cases 2 and 3 due to space restrictions and can be found in [3].

Node Allocation for Low Priority Requests: Figure 15 compares the average nodes allocated to the lower priority website during the burst of high priority requests for the prioritization and the prioritization with soft QoS schemes. We see that the prioritization with soft QoS scheme can provide a considerably better node allocation for the low priority requests as compared to the prioritization scheme in all three cases. It is to be noted that in spite of this better performance for low-priority requests, this scheme can meet the QoS guarantees of the high priority requests about 95% of

the times even with up to 50% lesser nodes as compared to a static allocation scheme.

We also evaluated the three models discussed above for the ZipF workload benchmark trace and a real-world WorldCup trace. We see similar trends in performance for the three models as shown in Figure 16.

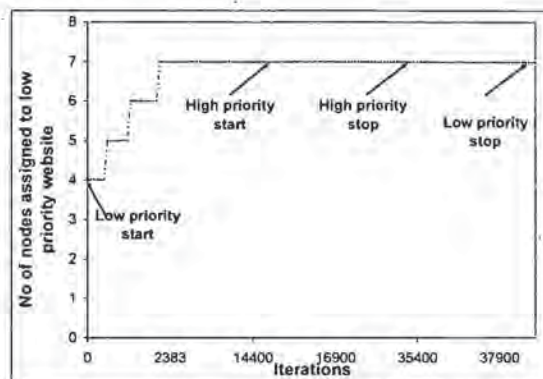


Figure 12. Basic Reconfigurability Node-Utilization

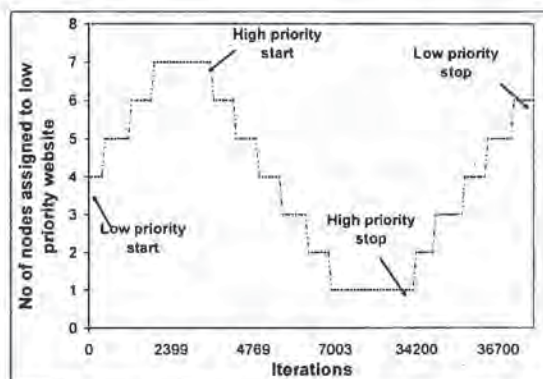


Figure 13. Prioritization Node-Utilization

6 Related Work

Several researchers have focused on the design of adaptive systems that can manage clusters and/or react to changing workloads in the context of web servers [19, 12, 24, 8, 22, 10, 13, 27]. There has been some previous research which focus on dynamism in the data-center environment by HP labs and IBM Research [17, 21]. These are notable in the sense that they were the first to show the capabilities of a dynamic allocation of system resources in the data-center environment. However, some of the solutions focus on lower level architectural requirements mainly for storage related

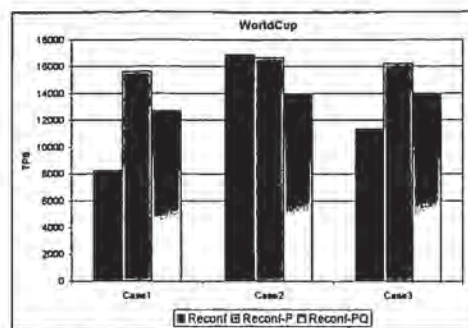
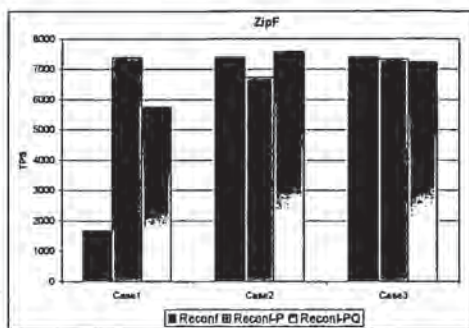


Figure 16. Performance for (a) Zipf and (b) WorldCup Traces

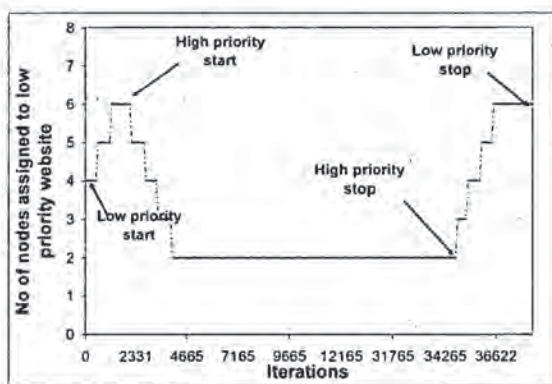


Figure 14. Soft QoS Node-Utilization

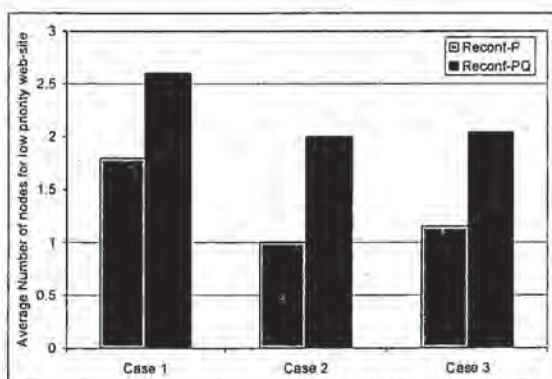


Figure 15. Average Number of nodes allotted to Low Priority Requests during a burst of high priority requests

issues, rely on specific hardware and are hard to look at as commodity component based solutions; other solutions require kernel-level modifications. On the other hand, in our approach, we try to propose a solution that is not geared toward any specific hardware and try to give a generic solution at the application level without requiring any changes to existing applications. Further, some of these approaches rely on the servers to intelligently reconfigure the other nodes. While these approaches are quite intuitive, in a real data-center scenario, the high server loads can make them inefficient and potentially unusable. Our approach of placing the onus of reconfigurability on the relatively lightly loaded edge servers by utilizing the remote memory operations offered by InfiniBand tries to tackle these challenges in an efficient manner.

Ranjan, et. al., have also previously looked at the problem of providing QoS guarantees in data-center environments [23]. However, this work does not deal with shared data-centers. Also, while the approach suggested in this paper are suitable and efficient for several data-centers, it might not be potentially usable for highly loaded or compute intensive data-centers due to the two sided nature of communication used.

Shah, Kim, Balaji, et. al., have done significant research in User Level High Performance Sockets implementations [26, 14, 15, 6, 7, 5]. In one of our previous works [5], we had evaluated the capabilities of such a pseudo-sockets layer over InfiniBand in the data-center environment. However, as we had observed in [20], the two-sided nature of Sockets API becomes an inherent bottleneck due to the high load conditions common in data-center environments. Due to this, we focused on the one-sided nature of InfiniBand to develop our external modules. Further, the existing data-center framework (Apache, PHP, etc.) is still based on the sockets API and can benefit from such high-performance sockets implementations. Thus, these approaches can be used in a complementary manner with our reconfigurability technique to make better utilization of system resources and provide high performance in a data-center environment.

7 Concluding Remarks

In the past few years several researchers have proposed and configured data-centers providing multiple independent services, known as shared data-centers. For example, several ISPs and other web service providers host multiple unrelated web-sites on their data-centers allowing potential differentiation in the service provided to each of them. Such differentiation becomes essential in several scenarios in a shared data-center environment.

In our previous work, we presented a novel design to provide dynamic reconfigurability of the nodes in the data-center environment. This technique enables the nodes in the data-center environment to efficiently adapt their functionality based on the system load and traffic pattern. However the basic dynamic reconfigurability scheme does not have concept of service differentiation *per se* and cannot be directly used in a shared data-center environment having different service requirements for different websites.

In this paper, we extended our basic dynamic reconfigurability scheme to allow service differentiation in the shared data-center environment. In particular, we pointed out the issues associated with the basic dynamic reconfigurability scheme and propose two extensions to it, namely (i) *Dynamic Reconfiguration with Prioritization* and (ii) *Dynamic Reconfiguration with Prioritization and Soft QoS*. We evaluated the benefits and issues associated with these schemes on an InfiniBand based cluster. Our experimental results showed that our extensions can allow the dynamic reconfigurability scheme to attain a performance improvement of up to 5 times for high priority websites irrespective of any background low priority requests. Further, our extensions also showed significant improvement in the performance of low priority requests when there are minimal or no high priority requests in the system. Finally, our extensions showed a similar performance as a static scheme with up to 50% lesser nodes in some cases.

We are currently working on multi-stage reconfigurations. In the scheme presented, the least loaded node reconfigures itself to belong to the highest loaded tier in an attempt to share the load. However, due to the heterogeneity (hardware components available) in the cluster, this might not be the optimal solution. On the other hand, a multi-stage reconfiguration, where a sequence of changes in the different tiers allowing the most appropriate node be reconfigured to the high load tier, could be more beneficial.

8 Acknowledgments

We would like to thank Savitha Krishnamoorthy and Dr. Jiesheng Wu for their contributions in the initial version of the reconfigurability scheme.

References

- [1] The Internet Traffic Archive. <http://ita.ee.lbl.gov/html/traces.html>.
- [2] InfiniBand Trade Association. <http://www.infinibandta.org>.
- [3] P. Balaji, S. Narravula, K. Vaidyanathan, H. W. Jin, and D. K. Panda. On the Provision of Prioritization and Soft QoS in Dynamically Reconfigurable Shared Data-Centers over InfiniBand. Technical report, The Ohio State University, Columbus, OH, Oct 2004.
- [4] P. Balaji, K. Vaidyanathan, S. Narravula, K. Savitha, H. W. Jin, and D. K. Panda. Exploiting Remote Memory Operations to Design Efficient Reconfiguration for Shared Data-Centers. In *Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations, and Technologies (RAIT)*, San Diego, CA, Sep 20 2004.
- [5] Pavan Balaji, Sundeep Narravula, Karthikeyan Vaidyanathan, Savitha Krishnamoorthy, Jiesheng Wu, and Dhabaleswar K. Panda. Sockets Direct Protocol over InfiniBand in Clusters: Is it Beneficial? In *the Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Austin, Texas, March 10-12 2004.
- [6] Pavan Balaji, Piyush Shivam, Pete Wyckoff, and Dhabaleswar K. Panda. High Performance User Level Sockets over Gigabit Ethernet. In *the Proceedings of the IEEE International Conference on Cluster Computing*, pages 179–186, Chicago, Illinois, September 23-26 2002.
- [7] Pavan Balaji, Jiesheng Wu, Tahsin Kurc, Umit Catalyurek, Dhabaleswar K. Panda, and Joel Saltz. Impact of High Performance Sockets on Data Intensive Applications. In *the Proceedings of the IEEE International Conference on High Performance Distributed Computing (HPDC)*, pages 24–33, Seattle, Washington, June 22-24 2003.
- [8] C. Lu and T. Abdelzaher and J. Stankovic and S. Son. A Feedback Control Approach for Guaranteeing Relative Delays in Web Servers. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, June 2001.
- [9] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic Resource Allocation for Shared Data Centers Using Online Measurements. In *Proceedings of ACM Sigmetrics 2003, San Diego, CA*, June 2003.
- [10] I-Hsin Chung and Jeffrey K. Hollingsworth. Automated Cluster-Based Web Service Performance Tuning. In *the Proceedings of the IEEE International Conference on High Performance Distributed Computing (HPDC)*, June 2004.
- [11] Voltaire Inc. <http://www.voltaire.com/>.
- [12] J. Carlstrom and R. Rom. Application-Aware Admission Control and Scheduling in Web Servers. In *Proceedings of the IEEE Infocom 2002*, June 2002.
- [13] A. Jacob, I. Troxel, and A. George. Distributed Configuration Management for Reconfigurable Cluster Computing. In *the Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, NV, June 21-24 2004.

- [14] Jin-Soo Kim, Kangho Kim, and Sung-In Jung. Building a High-Performance Communication Layer over Virtual Interface Architecture on Linux Clusters. In *the Proceedings of the IEEE International Conference on Supercomputing (ICS)*, pages 335–347, Naples, Italy, June 16-21 2001.
- [15] Jin-Soo Kim, Kangho Kim, and Sung-In Jung. SOVIA: A User-level Sockets Layer Over Virtual Interface Architecture. In *the Proceedings of the IEEE International Conference on Cluster Computing*, pages 399–408, California, USA, October 8-11 2001.
- [16] Cherkasova L. and Ponnkanti S. R. Optimizing a content-aware load balancing strategy for shared Web hosting service. In *8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 492 – 499, 29 Aug - 1 Sep 2000.
- [17] HP Labs. HP virtualization solutions: IT supply meets business demand: White Paper. In <http://h30046.www3.hp.com/uploads/infoworks/>, July.
- [18] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *ICS*, June 2003.
- [19] N. Bhatti and R. Friedrich. Web server support for tiered services. In *IEEE Network*, September 1999.
- [20] Sundeep Narravula, Pavan Balaji, Karthikeyan Vaidyanathan, Savitha Krishnamoorthy, Jiesheng Wu, and Dhabaleswar K. Panda. Supporting Strong Coherency for Active Caches in Multi-Tier Data-Centers over InfiniBand. In *the Proceedings of the IEEE International Workshop on System Area Networks (SAN)*, 2004.
- [21] Daniel OHare, Pankaj Tandon, Hemanth Kalluri, and Phil Mills. SNIA SSF Virtualization Demonstration. In *IBM Systems Group - TotalStorage Software: White Paper*, October.
- [22] P. Pradhan and R. Tewari and S. Sahu and A. Chandra and P. Shenoy. An Observation-based Approach Towards Self-Managing Web Servers. In *Proceedings of the Tenth International Workshop on Quality of Service (IWQoS 2002)*, May 2002.
- [23] S. Ranjan, J. Rolia, E. Knightly, and H. Fu. Qos-driven server migration for internet data centers, 2002.
- [24] S. Lee and J. Lui and D. Yau. Admission control and dynamic adaptation for a proportionaldelay diffserv-enabled web server. In *Proceedings of SIGMETRICS*, 2002.
- [25] Hemal V. Shah, Dave B. Mintum, Annie Foong, Gary L. McAlpine, Rajesh S. Madukkarumukumana, and Greg J. Regnier. CSP: A Novel System Architecture for Scalable Internet and Communication Services. In *the Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, pages pages 61–72, San Francisco, CA, March 2001.
- [26] Hemal V. Shah, Calton Pu, and Rajesh S. Madukkarumukumana. High Performance Sockets and RPC over Virtual Interface (VI) Architecture. In *the Proceedings of the CANPC workshop (held in conjunction with HPCA Conference)*, pages 91–107, 1999.
- [27] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated Resource Management for Cluster-based Internet Services. In *Proceedings of Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 225–238, Boston, MA, December 2002.
- [28] J. Wu, P. Wyckoff, and D. K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. In *ICPP*, 2003.
- [29] George Kingsley Zipf. Human Behavior and the Principle of Least Effort. Addison-Wesley Press, 1949.



THE POWER OF INFINITE

Powered by SCoze

The RIKEN Super Combined Cluster and SCoze Cluster System Software

-- Achieves 8.72 TFLOPS
on Cluster of Clusters --

Shinji Sumimoto
Fujitsu Laboratories, Japan

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004



THE POWER OF INFINITE

The RIKEN Cluster is #7 on TOP500

24th List (2004.June)

#	System name, Processor, Network	Rmax (GFlops)	NProc	Type
1	Earth Simulator	35,860	5,120	Vector
2	Thunder – Itanium2, 1.4 GHz	13,880	8,192	PC Cluster
3	ASCI Q – AlphaServer SC45, 1.25 GHz	10,280	2,200	Scalar
4	BlueGene LDD1 Prototype, PPC 440 0.5GHz	11,680	8,192	Scalar
5	Tungsten, Xeon 3.06 GHz, Myrinet	9,819	2,500	PC Cluster
6	eServer p690, Power4+ 1.9GHz	8,955	2,112	Scalar
7	RIKEN Super Combined Cluster, Xeon 3.06, (InfiniBand(FJ+Topspin)+MyrinetXP) GigaE	8,728	2,048	PC Cluster
8	BlueGene LDD2 Prototype, PPC 440 0.7GHz	8,655	4,096	Scalar
9	MPP2, Itanium2 1.5GHz, Quadrics	8,633	1,936	PC Cluster
10	Dawning4000A Opteron 2.2GHz, Myrinet	8,061	2,560	PC Cluster

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004

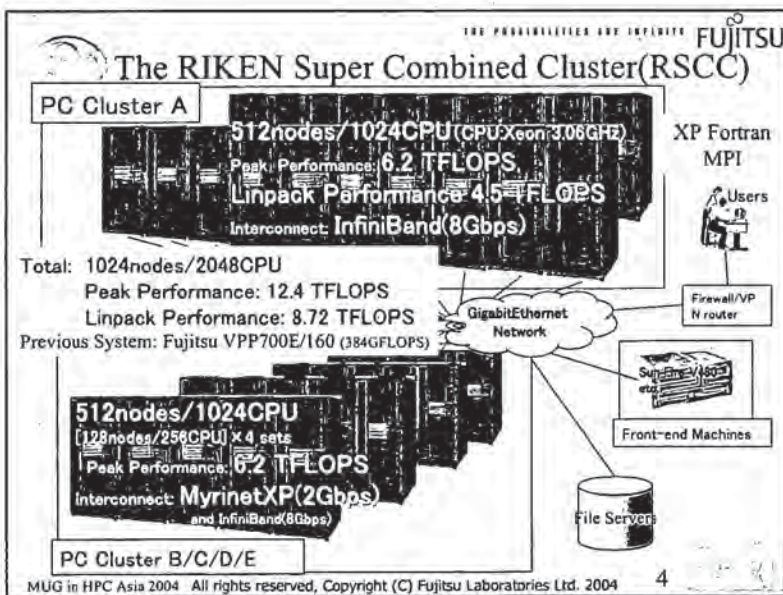
THE POSSIBLE IS THE INFINITE **FUJITSU**

Comments from TOP500 Highlights of 24th List

- The RIKEN Super Combined Cluster(RSCC):
 - “#7 is a second Japanese system build by Fujitsu and installed at RIKEN. It is special as it is a heterogeneous cluster with nodes of different capabilities which make it more challenging to achieve good performance. It is the first such system that high in the top500.”
- The RSCC Cluster is configured as Cluster of Clusters.

3

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004





THE POSSIBILITIES ARE INFINITE
Outline of This Talk



- Challenges of RSCC
- The Solution for the Challenge
- Linpack Measurement for Top500

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004
5




THE POSSIBILITIES ARE INFINITE
Challenges of the RSCC System



- Using PC Cluster for Computer Center
 - Requires the same functionality and reliability as VPP as much as possible.
 - High Application Performance
- Usability on Multiple Clusters
 - Program binary compatibility
- Migration of VPP Applications to PC Cluster
 - XP Fortran Programs of Users



MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004
6


THE POSSIBILITIES ARE INFINITE **FUJITSU**



SCore is the Solution

- Use of PC Cluster for Computer Center
 - NQS(Network Queuing System) and NQS-JM with relationship to Checkpoint-restart and Job Freeze on SCore using InfiniBand and Myrinet
- Usability on Multiple Clusters
 - A single binary can run on both InfiniBand and Myrinet Cluster on SCore.
- Migration of VPP Applications to PC Cluster
 - Providing of XP Fortran for SCore

7

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004

THE POSSIBILITIES ARE INFINITE **FUJITSU**



Fujitsu's NQS Batch Queuing System

- NQS: Batch Queuing system with same usability as same as VPP NQS
 - With Job Freeze Interface for SCore
 - File staging support using pre-post script
- NQS-JM: Job Manager for NQS with load balancing facilities.
 - Detecting Node down and maintaining load balancing

8

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004

THE FUNDAMENTALS AND JOURNALS FUJITSU

Fujitsu's NQS Batch System on SCore

- NQS daemon schedules jobs using several queues in which priority and limitation of number of nodes.
- A user submits a job script.
- The NQS decides where a job is executed in a cluster. Then NQS runs the job using scrun of SCore

Control Node Computational Nodes

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 9

THE FUNDAMENTALS AND JOURNALS FUJITSU

Checkpoint on SCore

- To use checkpoint on SCore:
 - For Interactive Jobs: Specifies checkpoint option on scrun
 - For NQS Jobs: Qsub command adds the checkpoint option.

Management Node Computational Nodes

Checkpoint File

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 10

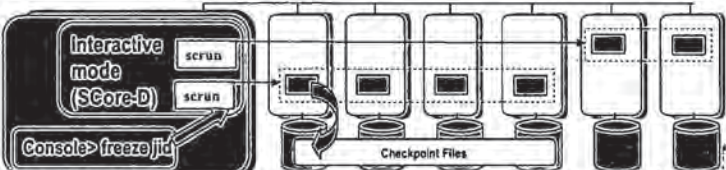


THE PARALLELIZATION AND INTERPRET

FUJITSU

Job Freeze Function on SCore

- System administrator can make checkpoint file of specified job(s) using freeze command on sc_console.
- `Console> freeze <IID>|"all"`
- After freeze command is finished, the system can be stopped.
- The system can be restarted
 - % `scrun -reconnect=<ID>, ./a.out`
 - % `scored -restart`



MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004



THE PARALLELIZATION AND INTERPRET

FUJITSU

XP Fortran

- Automatic Parallelized Fortran Compiler using directives
 - Originally developed for NWT(Numerical Wind Tunnel) which was TOP 1 machine Nov. 1993 in Top500 List.
- Supported Platform:
 - VPP(UXP/M,V)
 - PRIMEPOWER(Solaris)
 - PRIMERGY(IA32-Linux)



MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004



THE POSSIBILITIES ARE INFINITE

XP Fortran on SCore



- Language Specification
 - Virtual Global Address Space
 - Procedure distribution
 - Data Movement
- XP Fortran on SCore
 - Implemented using PMv2 interface directly
 - HIMENO Benchmark Results
 - 1 CPU: 812 MFLOPS, 16CPU: 10,164 MFLOPS using PM/InfiniBand-FJ

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004

13



THE POSSIBILITIES ARE INFINITE

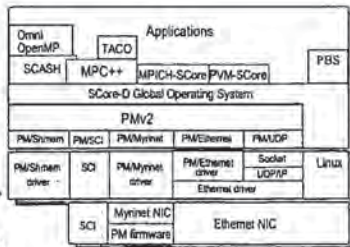
SCore Cluster System Software

<http://www.pccluster.org>



Developed by RWC Project(1992-2002)

- Now, PC Cluster Consortium develops and supports SCore
- PMv2: High Performance Communication Facility
 - Multiple types of network support on single API: Myrinet, Ethernet, Shmem, SCI, InfiniBand, RHiNET
- SCore-D: Global Operating System
 - Support checkpoint-restart functionality
 - Realizing seamless cluster environment using PM API



MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004

14



THE FUNDAMENTALS AND INFRASTRUCTURE

PMv2



- Developed to Eliminating Existing Protocol Processing Overhead
- Supporting multiple types of cluster interconnect
 - PM/Myrinet, PM/Ethernet, PM/Shmem, PM/SCI, PM/UDP(Agent), PM/Infiniband (-FJ[Fujitsu], -TS[TopSpin])
- Providing Seamless Network Communication Environment
 - PM API is defined to provide PM feature for device specific PM device.
 - PM/Composite is used to allocate network device on job submission using scrum command dynamically.

PMv2: High Performance Communication Facility

PM/Shmem	PM/Myrinet	PM/InfiniBand	PM/Ethernet	PM/Composite
PM/Shmem Driver	PM/Myrinet Driver	PM/InfiniBand Driver	PM/Ethernet Driver Ethernet Driver	Linux
	Myrinet NIC	IB HCA	Ethernet NIC	
	PM Firmware	IB Firmware		

MUG in HPC Asia 2004 All rights reserved, Co.



THE FUNDAMENTALS AND INFRASTRUCTURE

PMv2 Technologies for High Performance Communication



- Reliable Ordered Datagram Communication: PMv2
 - Connection Oriented Communication is not suited for Large Scale Cluster
- User Level Communication: PM/Myrinet, PM/InfiniBand
 - Eliminating System Call Overhead for Low Latency Network
- Zero-Copy Communication: PM/Myrinet, PM/InfiniBand
 - Eliminating CPU Copy Overhead to Increase Communication Bandwidth
- Network Trunking Technique: PM/Ethernet
 - Increasing Communication Bandwidth using Multiple Network
- Kernel Level RMA: PM/Ethernet-kRMA
 - A new approach to realize high performance communication

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004

THE POSSIBILITIES ARE INFINITE **FUJITSU**

PM/Myrinet for Myrinet Interconnect

- PM Communication Protocol is realized in LANai processor.
 - User Level Communication
 - Zero-Copy Communication
- Reliable Protocol for Myrinet2000 with Serial/Fiber link for CRC error and Packet Lost
- Performance
 - Myrinet 2000(PCI64B)
 - Bandwidth: 240MB/s
 - RTT Latency: 11 usec
 - Myrinet XP
 - Bandwidth: 243 MB/s
 - RTT Latency: 8.3 usec

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 17

THE POSSIBILITIES ARE INFINITE **FUJITSU**

PM/Ethernet for Commodity Ethernet

- Implemented on Linux with minimal kernel modification
 - Using existing device driver interface
 - No modification to existing Ethernet device driver
 - Light weight network protocol is implemented on the PM/Ethernet device driver
- Modification to the Linux kernel
 - Ethernet frame handler on Data Link Layer
 - Registering interrupt handler information

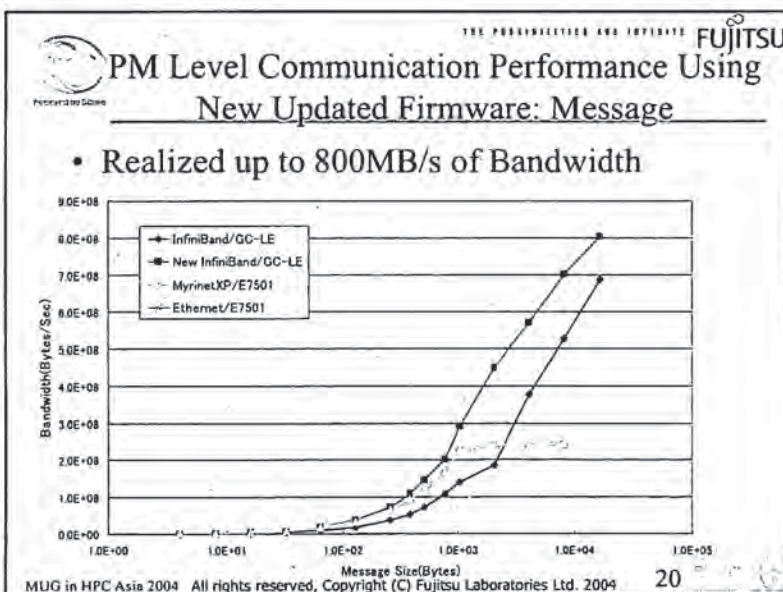
MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 18

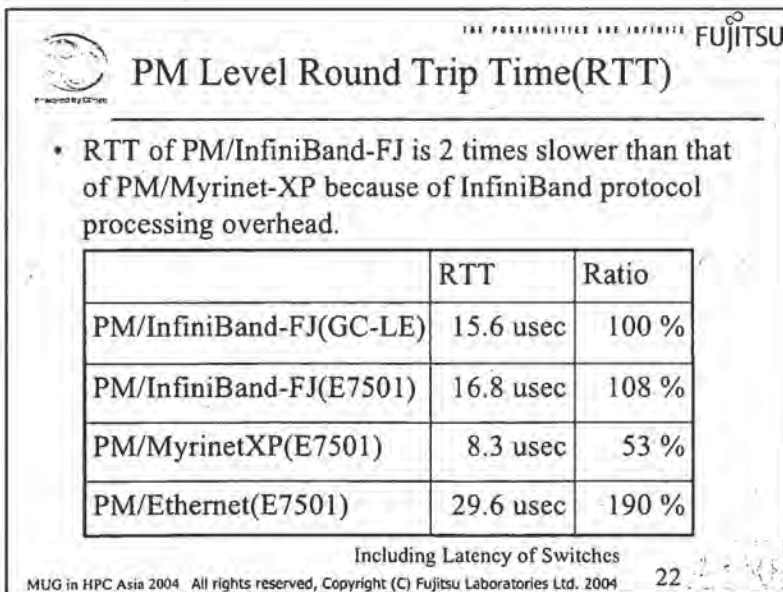
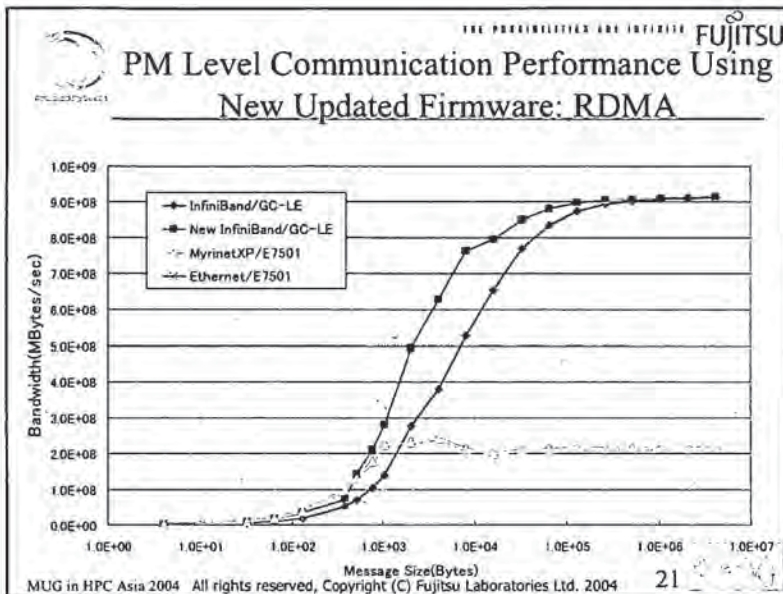
THE PERFORMANCE AND INFINITE **FUJITSU**

PM/InfiniBand-FJ for Fujitsu HCA

- Reliable Connection Queue Pair is Used
 - User Level Communication
 - Zero-Copy Communication
- Realizing high communication performance and high reliability by extending the specifications of InfiniBand
 - Introducing ASQP-SROP to reduce receive buffer consumption
 - Reducing communication latency by eliminating number of PCI DMA
 - Realizing Checkpoint-restart function by using SCore

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 19

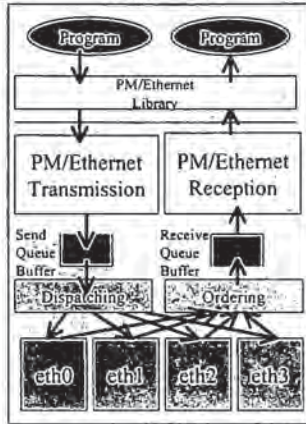




THE POSSIBLES ARE INFINITE FUJITSU

Network Trunking on PM/Ethernet

- Providing Higher Network Communication Bandwidth Using Multiple Networks
 - Providing Message Model.
- Implementation:
 - Using Ethernet MAC addresses of other PM/Ethernet units.
 - On message transmission: selects a NIC, and replaces Ethernet MAC address and submit the message to the NIC's driver.
 - On message reception: introduces a receive message queue for message ordering.



The diagram illustrates the network trunking architecture. At the top, two 'Program' boxes are connected to a 'PM/Ethernet Library'. Below the library, there are two main paths: 'PM/Ethernet Transmission' and 'PM/Ethernet Reception'. The transmission path includes a 'Send Queue Buffer' and a 'Dispatching' block. The reception path includes a 'Receive Queue Buffer' and an 'Ordering' block. Both paths lead to a set of four network interface cards labeled 'eth0', 'eth1', 'eth2', and 'eth3'. Arrows indicate the flow of data from the programs through the library and buffers to the specific NICs for transmission, and from the NICs through the ordering block and receive buffer back to the library for reception.

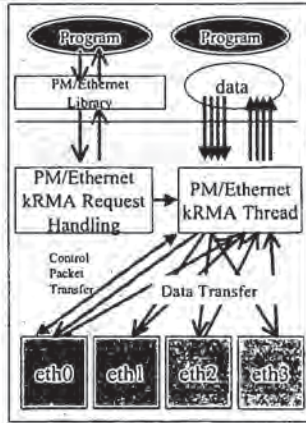
23

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004

THE POSSIBLES ARE INFINITE FUJITSU

PM/Ethernet-kRMA

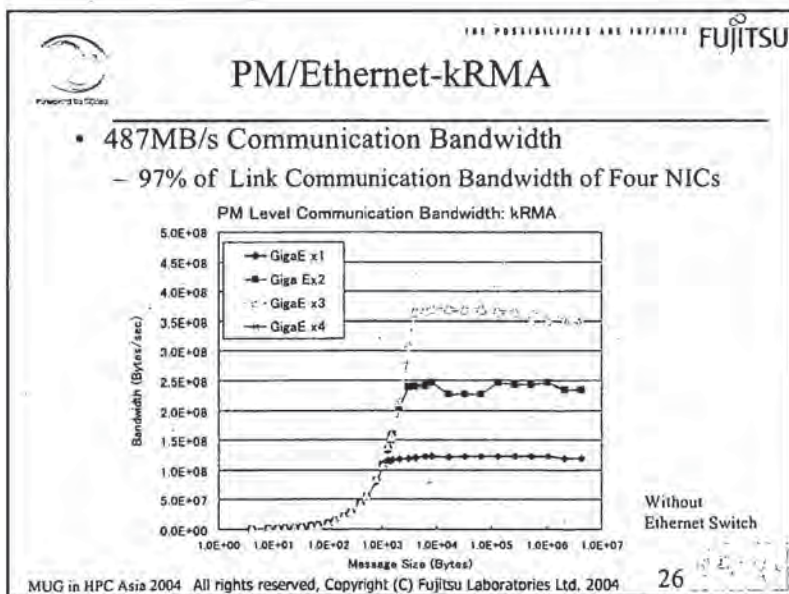
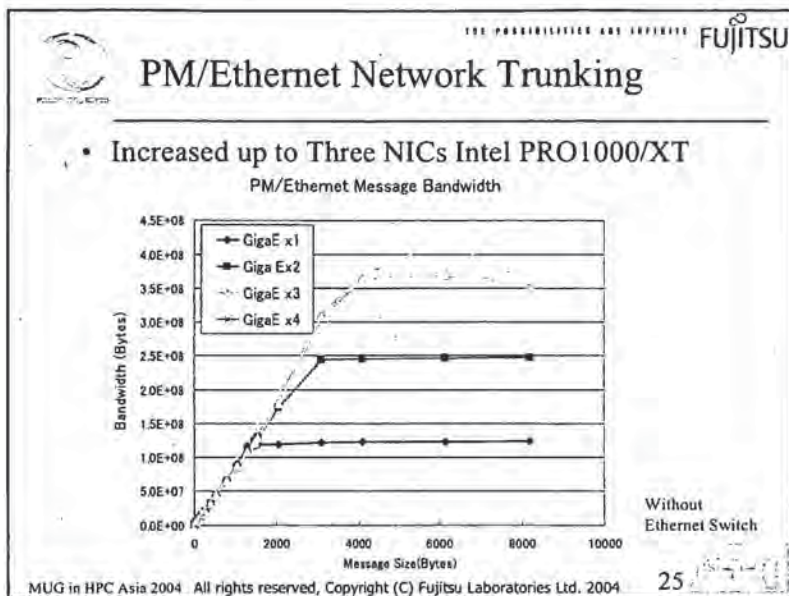
- Kernel Level Remote Memory Access Facility Using Multiple Gigabit Ethernet Cards
 - One CPU Copy Communication
 - Accessing using process virtual address
- RMA Processing is processed on kernel.
 - No send buffer and receive buffer
 - No dispatching and ordering function



The diagram illustrates the PM/Ethernet-kRMA architecture. At the top, two 'Program' boxes are connected to a 'PM/Ethernet Library'. Below the library, there are two main components: 'PM/Ethernet kRMA Request Handling' and 'PM/Ethernet kRMA Thread'. A 'data' box is shown with multiple arrows pointing to the 'kRMA Thread'. The 'kRMA Thread' is connected to a set of four network interface cards labeled 'eth0', 'eth1', 'eth2', and 'eth3'. Arrows indicate the flow of 'Control Packet Transfer' and 'Data Transfer' between the thread and the NICs.

24

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004



THE FUNDAMENTALS ARE INFINITE FUJITSU

PM/Ethernet Experience on 10Gbit Ethernet

- Hardware:
 - PRIMERGY RX200 (Dual Xeon 3.06GHz)
 - 10GE NIC: Intel PRO/10GbE LR Server Adapter
 - Switch: XG-800 based on 12 port 10GE Switch LSI
 - One chip 12 port LSI: 460ns Latency
- PM Level Performance:
 - Message Bandwidth: 773MB/s
 - Round Trip Time: 17.6 usec




4 Node Cluster

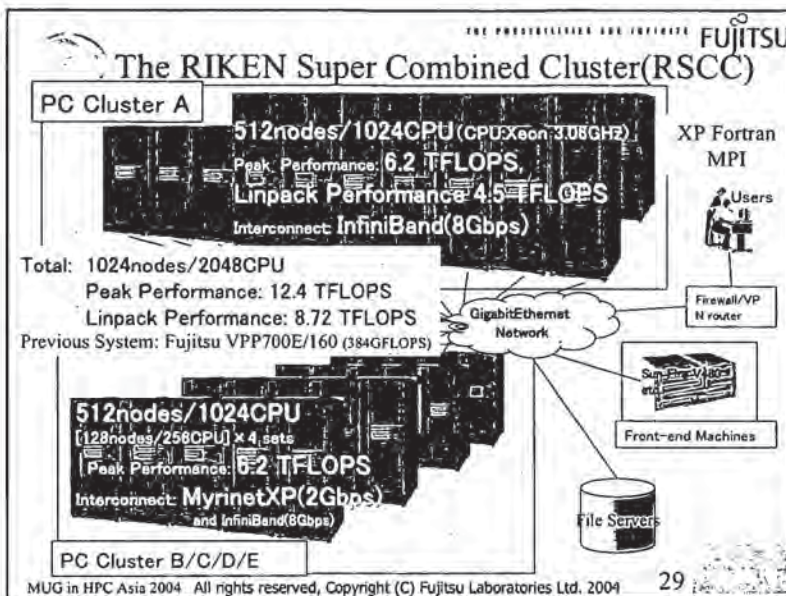
MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 27

THE FUNDAMENTALS ARE INFINITE FUJITSU

Linpack Measurement on RSCC

- Configuration of the RSCC
- Our challenges of Linpack for Cluster of Clusters
 - PM/Composite
 - FLAB DGEMM
 - HPL Tuning
- Measurement of Linpack

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 28



THE POSSIBILITIES ARE INFINITE **FUJITSU**

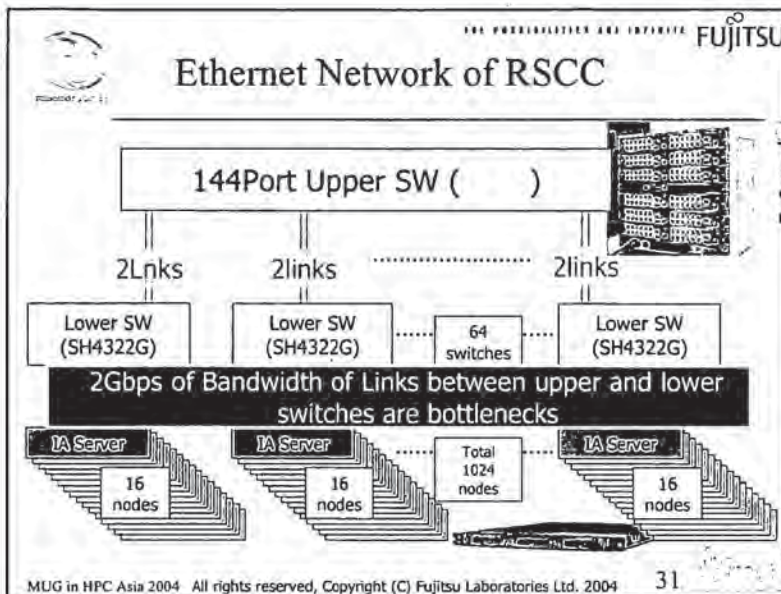
RSCC Configuration

- RSCC is a Cluster of 5 Clusters

Cluster Name	# Nodes/# CPUs	Memory/Node	Interconnect
PC1	512 / 1024	4GB	IB (FJ)
PC2A	128 / 256	2GB	MyrinetXP
PC2B	128 / 256	2GB	MyrinetXP
PC2C	128 / 256	2GB	MyrinetXP
PC2D	128 / 256	2GB	IB (TopSpin, Mellanox)

- All of the clusters are connected by Gigabit Ethernet

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 30



THE POSSIBILITIES ARE INFINITE. FUJITSU

Motivation of Using Cluster of Clusters

- 512 node InfiniBand Cluster only achieved 4.5 TFLOPS in spite of total 12.5 TFLOPS.
- We started to increase performance using whole of 1024 node cluster.
- However, Gigabit Ethernet can only be used as cluster network that covers whole of nodes.
- Therefore, we try to use cluster network as much as possible to reduce traffic of Ethernet network.
 - Change network allocation using PM/Composite
 - HPL modification

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 32

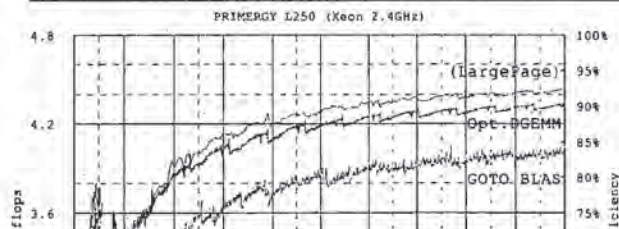


Our Approach

- Achieving Higher DGEMM Performance on Single Processor
 - Developing Fastest DGEMM on Xeon
- Minimizing the Utilization of Ethernet Network by Maximizing Utilization of Interconnect
 - PM/Composite
 - HPL Tuning



DGEMM Performance Comparison



Performance on Xeon 2.4GHz	Average		Peak	
	GFlops	Efficiency	GFlops	Efficiency
GOTO DGEMM	3.69	76.8%	4.06	84.5%
FLAB-DGEMM	4.04	84.2%	4.34	90.4%
(large page)	4.12	85.8%	4.43	92.4%

THE POSSIBLE IS THE ONLY LIMIT **FUJITSU**

FLAB DGEMM vs. GOTO DGEMM

- Leveling of FSB load Using Software Pre-fetch

- Blocksize of Core routine
 - GOTO DGEMM: 112 / FLAB DGEMM: 156

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 35

THE POSSIBLE IS THE ONLY LIMIT **FUJITSU**

Increasing Communication Performance on RSCC

- 2 type of Network
 - Computation Network: IB, Myrinet
 - High Communication Performance, but limited coverage
 - Management Network: GbE
 - Low Communication Performance, but covers all nodes
- Shifting communication load to computation network
 - PM/Composite (SCore) Communication Facility
 - Selecting PM network for each CPU
 - shmem > IB or Myrinet > GbE
 - No need for changing application programs
 - A new Node mapping is developed using HPL

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 36



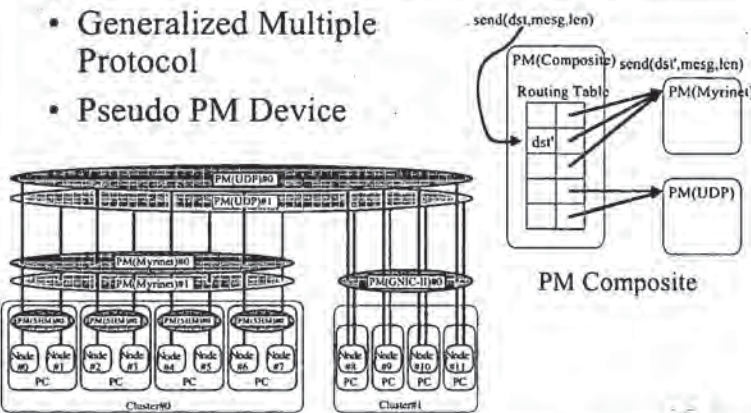
PM/Composite

THE PARALLELIZATION AND INFINITE

FUJITSU

SCore 3.0
New Feature

- Generalized Multiple Protocol
- Pseudo PM Device



MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 37



PM/Composite for RSICC

THE PARALLELIZATION AND INFINITE

FUJITSU

- PM device is changed by each destination node

PM/Shmem is used within a node

PM/InfiniBand-FJ	PM/Myrinet	XP	PM/InfiniBand-TS
PC Cluster A	B	C	D
E			
PM/Ethernet			

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 38

THE POSSIBILITIES ARE INFINITE FUJITSU

Linpack using HPL

Update

0	8	16	24	32	40	48	56
1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63

NB

N

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004

39

THE POSSIBILITIES ARE INFINITE FUJITSU

HPL Node Allocation to Shift Communication Load to Faster Network

- Node Mapping:
 - HPL: Two Kind of Node Mapping
 - Y Axis Priority Mapping
 - X Axis Priority Mapping

Y Axis

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

X Axis

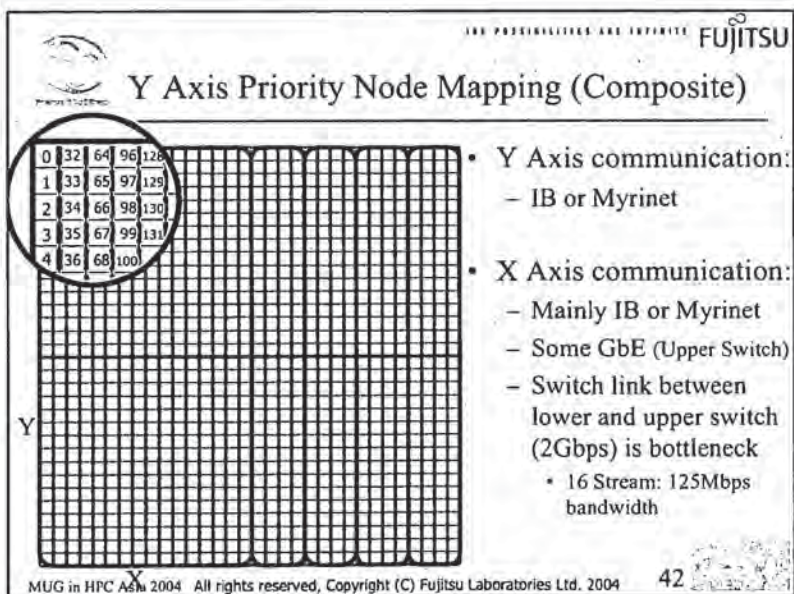
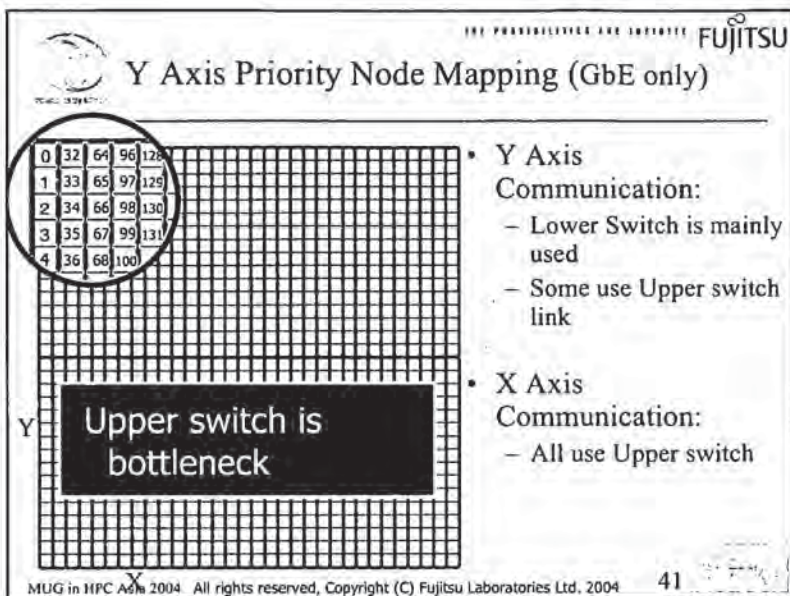
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

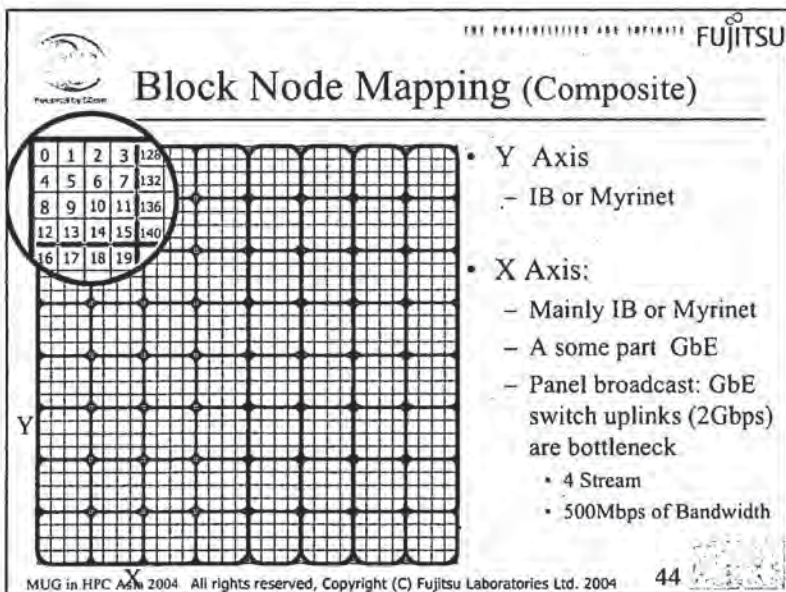
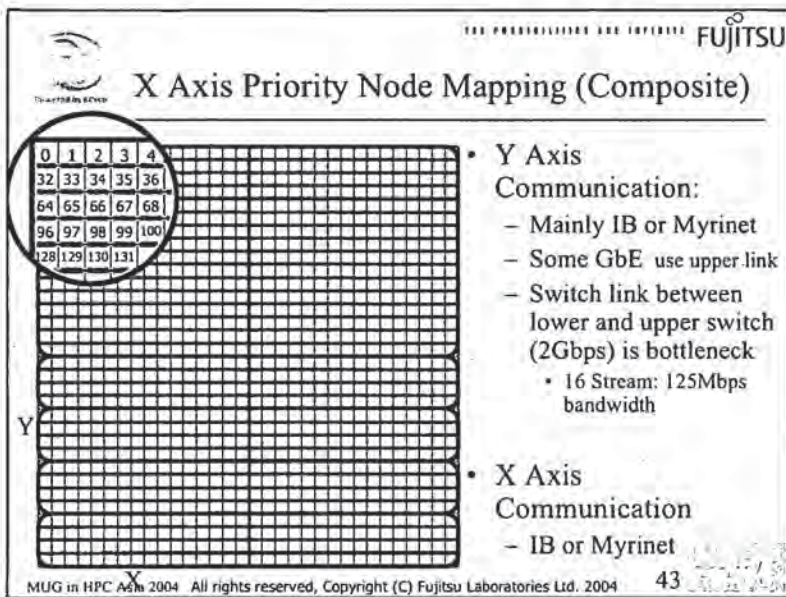
EX: Node Mapping
(16 Nodes Grid:(4,4))

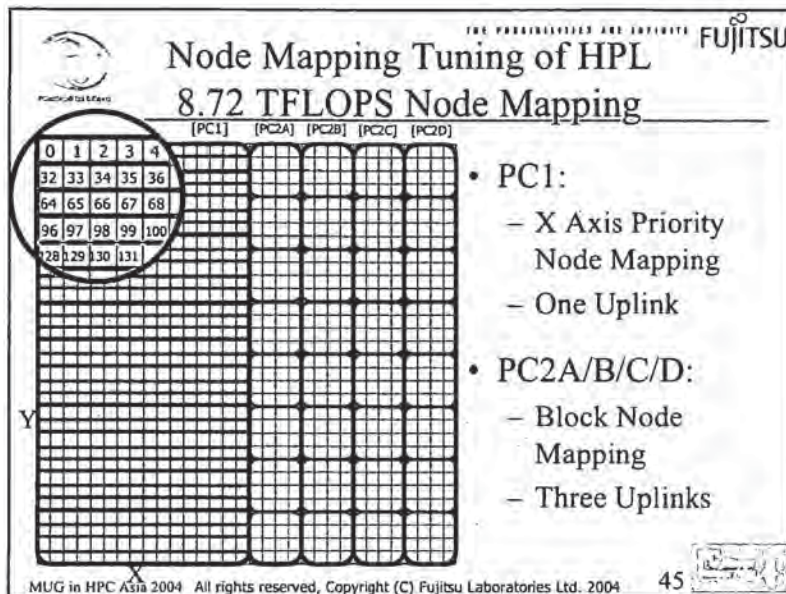
- Optimized Node Mapping of RSCC
 - 1024 Nodes, Grid:(32,32)

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004

40







THE POSSIBLE IS THE INFINITE **FUJITSU**

Summary

- The RIKEN Super Combined Cluster : Challenge
 - Using PC Cluster for Computer Center
 - Usability on Multiple Clusters
 - Migration of VPP Applications to PC Cluster
- SCore Cluster System Software is the Solution
 - NQS and XP Fortran
 - Job Freeze Function
- Linpack Measurement
 - Achieved 8.72 TFLOPS using Fastest DGEMM, PM/Composite and a new HPL program.

MUG in HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004

46



THE POSSIBILITIES ARE INFINITE **FUJITSU**

Powered by SCore

PM/InfiniBand-FJ: A Design of High Performance Communication Facility Using InfiniBand for Large Scale PC Clusters

Shinji Sumimoto, Akira Naruse, Kouichi Kumon
(Fujitsu Laboratories, Japan)

Kouji Hosoe, Toshiyuki Shimizu (Fujitsu, Japan)

HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 1



THE POSSIBILITIES ARE INFINITE **FUJITSU**

Powered by SCore

Background

- PC Cluster has become a major computing platform in High Performance Computing Area.
 - Cluster interconnect (ex InfiniBand and 10Gbit Ethernet) has become comparable in communication performance to that of commercial super computer.
 - Many 1000 node clusters have been built.
- However, more system operability and availability are needed for PC clusters in order to replace commercial traditional super computer.
 - Checkpoint-Restart Function for long period jobs, and relationship with batch system
- PC Cluster with 10Gbps Class Network + High Availability will increase PC Cluster user for business use.

Therefore, we started to develop communication facility:
PM/InfiniBand-FJ using InfiniBand with high availability

HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 2



THE POSSIBLE IS AN INFINITE

Outline of This Talk

- The goal of developing PM/InfiniBand-FJ
- What is InfiniBand?
- Design of PM/InfiniBand-FJ
- Implementation
- Evaluation

HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 3



THE POSSIBLE IS AN INFINITE

Goals of Developing PM/InfiniBand-FJ

- High Communication Performance:
 - Bringing communication performance close to its 10Gbps class physical link performance
- High Scalability:
 - Realizing maximum application performance using minimum consumption of computer resources on 1000 node cluster
- High Availability:
 - Realizing checkpoint and restart feature for system operation

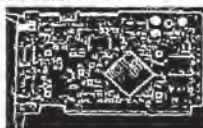
HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 4



What is InfiniBand?

- High Performance Interconnect with 1x(2Gbps), 4x(8Gbps), 12x(24Gbps)
 - Specified in InfiniBand(SM) Trade Association
<http://www.infinibandta.org>
 - API is not defined, VERB is defined
- Connection Oriented, Datagram and RAW Comm.
 - Reliable and Un-reliable
 - Send(Recv), RDMA(Read/Write), Atomic Operation

■ ICA Card

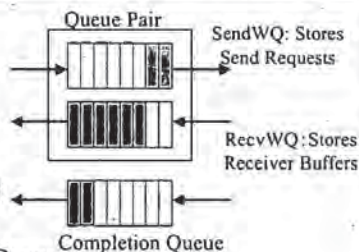


■ 32 port switch by InfiniCon



Communication Using InfiniBand

- QueuePair(QP) is used
 - A request is posted to WQ(WorkQueue) : Send Request, Receive Buffer Post Request...
 - A Receiver buffer which size is MTU is posted to WQ
- Completion of request is notified using Completion Queue
- Queue Operation is performed to change QP state: Create(), Destroy(), etc..
- Reliable communication can be used for Cluster Communication





Reliable Connection vs. Reliable Datagram

THE POSSIBILITIES ARE INFINITE FUJITSU

- RC (Reliable Connection) Communication
 - Connection Oriented Communication: Requires Number of QPs same as number of destinations
 - Outstanding Requests can be performed
- RD (Reliable Datagram) Communication
 - Datagram Communication: Only one QP is Required for communicating to all other destinations
 - Outstanding Request is not permitted
- RC Communication is an only choice for High Performance Communication
 - A number of QP resources such as receive buffer are required on Large Scale Cluster

Reduction of QP resources is an issue of RC communication

HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004



Design Policies of PM/InfiniBand-FJ

THE POSSIBILITIES ARE INFINITE FUJITSU

- Fujitsu's InfiniBand HCA (IB HCA) is used.
 - The specification of InfiniBand can be changed when needed.
 - However, specification changes and modifications of IB HCA should be minimized.
- SCore Cluster System Software is used.
 - Developing PMv2 Communication Facility for IB HCA
 - Realizing High Performance Communication and Checkpoint-Restart

HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004

8



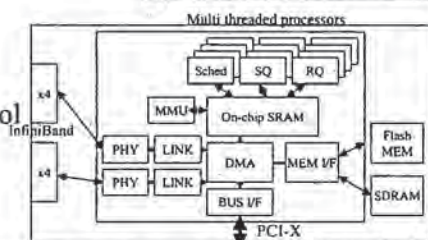


Fujitsu's InfiniBand HCA (IB HCA)

THE POSSIBILITIES ARE INFINITE

FUJITSU

- InfiniBand 4x Spec.
- PCI-X 133MHz
- Multi-threaded Processors
 - Processor Clock 266MHz
 - Send, Recv, Sched
- Some part of communication protocol is processed on firmware



HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004

9



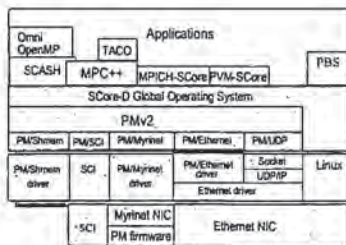
SCore Cluster System Software

<http://www.pccluster.org>

THE POSSIBILITIES ARE INFINITE

FUJITSU

- PMv2: High Performance Communication Facility
 - Multiple types of network support
 - Realizing seamless cluster environment using PM API
- SCore-D: Global Operating System
 - Support checkpoint-restart functionality



HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004

10



THE PRIORITIES ARE RESERVED
FUJITSU

Design Issues of PM/InfiniBand-FJ

- Realizing High Communication Performance:
 - High Bandwidth and Low Latency Communication
- Realizing High Scalability:
 - A receive buffer providing method which is independent of number of QPs
- Realizing High Availability:
 - Checkpoint-restart
- We discuss the issue “Realizing High Scalability”

HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004


11



THE PRIORITIES ARE RESERVED
FUJITSU

Receive Buffer Problem on RC Communication

- Required about 1GB of memory on 1024 Node Cluster
 - Entries of RecvWQ: 128, MTU 8KB: 1MB/QP
- 1GB of memory usage impacts on 32bit processor systems
- We solve the problem by introducing ASQP-SRQP: Sharing receive buffer queue for multiple QPs.

HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004


12

THE POSSIBILITIES ARE INFINITE FUJITSU

Sharing Receive Buffer Queue: ASQP-SRQP

- SRQP: Give Receive Buffer queue independence from RCQP

HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 13

THE POSSIBILITIES ARE INFINITE FUJITSU

Design Decisions of PM/InfiniBand-FJ

- High Communication Performance:
 - High Bandwidth: RDMA Communication,
 - Low Latency: Eliminating Number of PCI DMAs
- High Scalability:
 - Introducing the new ASQP-SRQP to reduce consumptions of receive buffer
- High Availability:
 - Adding Interface of saving and restoring of QP context data on IB HCA

We have realized these feature by extending the specifications of InfiniBand

HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 14

THE POSSIBILITIES ARE INFINITE **FUJITSU**

Implementation of PM/InfiniBand-FJ

- Consists of User Library, Device Driver, and IB HCA Firmware
 - Device Driver: Creating and Destroying QPs, Controlling IB HCA
 - User Library: Message sending and receiving using QPs
 - IB HCA Firmware: Extensions of InfiniBand Specification

HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 15

THE POSSIBILITIES ARE INFINITE **FUJITSU**

Implementation of PM/InfiniBand-FJ(2)

Firmware Extensions of IB HCA

- Extending the IB Specification to realize ASQP-SRQP
- Checkpoint feature is only supported on ASQP-SRQP
- Multiple QPs can be operated by single operation.
- Lower Link Protocol does not changed.
- Original QPs (RCQP, UDQP) can be also used.

HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 16



THE POSSIBILITIES ARE INFINITE
FUJITSU

Comparison of Receiver Buffer Usage

- Receive Buffers on ASQP-SRQP: 8MB
 - MTU 16KB, 512 Entries, 1024 Nodes
- Receive Buffers on RCQP: 2GB
 - MTU 16KB, 128 Entries, 1024 Nodes

Receive Buffer Usage on ASQP-SRQP is 1/256 of that on RCQP

HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004

17




THE POSSIBILITIES ARE INFINITE
FUJITSU

Performance Evaluation

- Comparison of PM Level Communication Performance with InfiniBand, MyrinetXP and Gigabit Ethernet.
 - Round Trip Time
 - Message Communication Bandwidth
 - RDMA Communication Bandwidth
- Application Performance
 - NPB 2.3 IS Class B: Compared with InfiniBand, MyrinetXP and Gigabit Ethernet

HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004

18




THE POSSIBILITIES ARE INFINITE

FUJITSU

Evaluation Environment

- Communication Performance
 - Self-made PC Cluster: SCore 5.6.1
 - Xeon 2.8GHz ServerWorks GC-LE Chipset, PCI-X 133MHz, 1GB Memory, InfiniBand
 - PRIMERGY RX200 Cluster: SCore 5.6.1
 - Xeon 3.06GHz Intel E7501 Chipset, PCI-X 133Mz, 2GB Memory, InfiniBand, MyrinetXP, Gigabit Ethernet
- Application Performance
 - PRIMERGY RX200 Cluster
256 Nodes(InfiniBand), 128 Nodes(MyrinetXP)

HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004

19

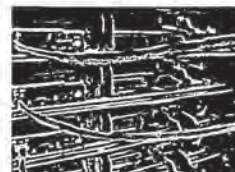
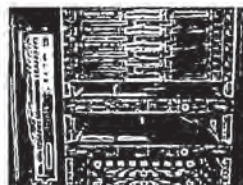


THE POSSIBILITIES ARE INFINITE

FUJITSU

PRIMERGY RX200 Cluster

- PRIMERGY RX200 (Xeon 3.06GHz, E7501)



HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004

20



PM Level Round Trip Time(RTT)

- RTT of PM/InfiniBand-FJ is 2 times slower than that of PM/Myrinet-XP because of InfiniBand protocol processing overhead.

	RTT	Ratio
PM/InfiniBand-FJ(GC-LE)	15.6 usec	100 %
PM/InfiniBand-FJ(E7501)	16.8 usec	108 %
PM/MyrinetXP(E7501)	8.3 usec	53 %
PM/Ethernet(E7501)	29.6 usec	190 %

Including Latency of Switches

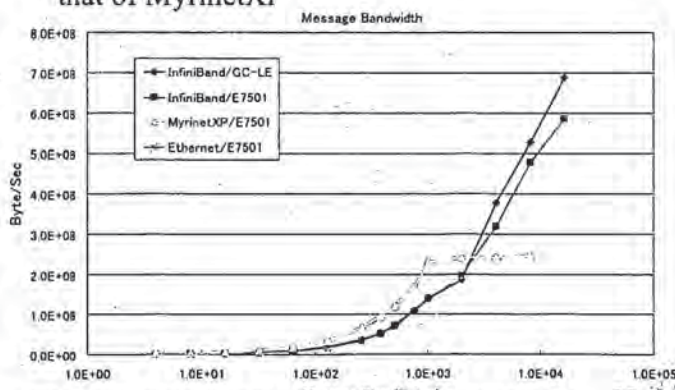
HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004

21



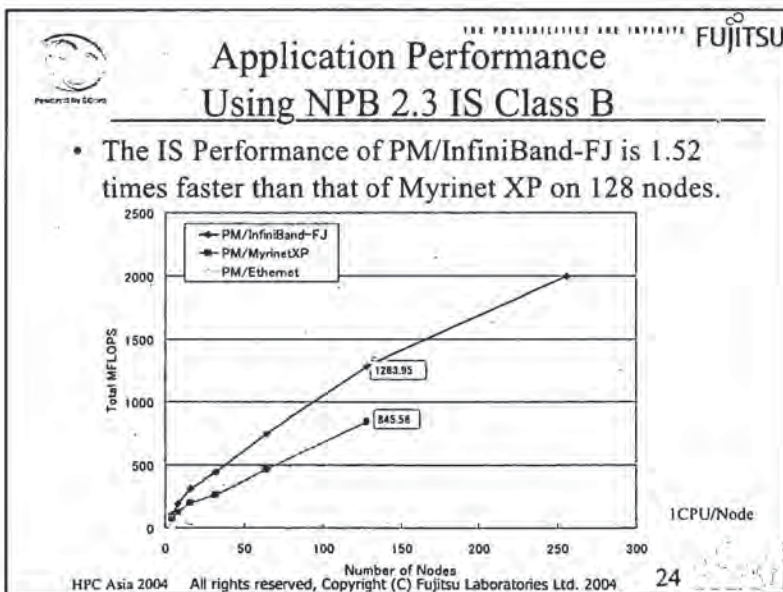
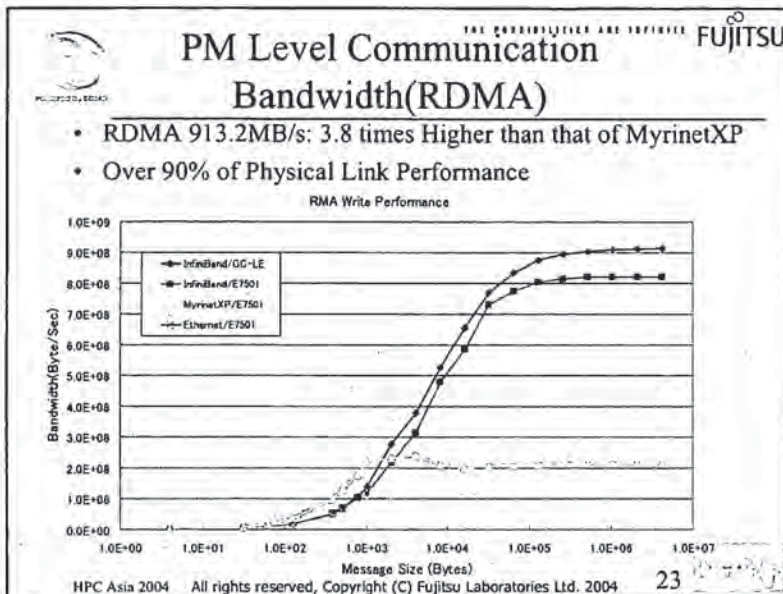
PM Level Communication Bandwidth(Message)

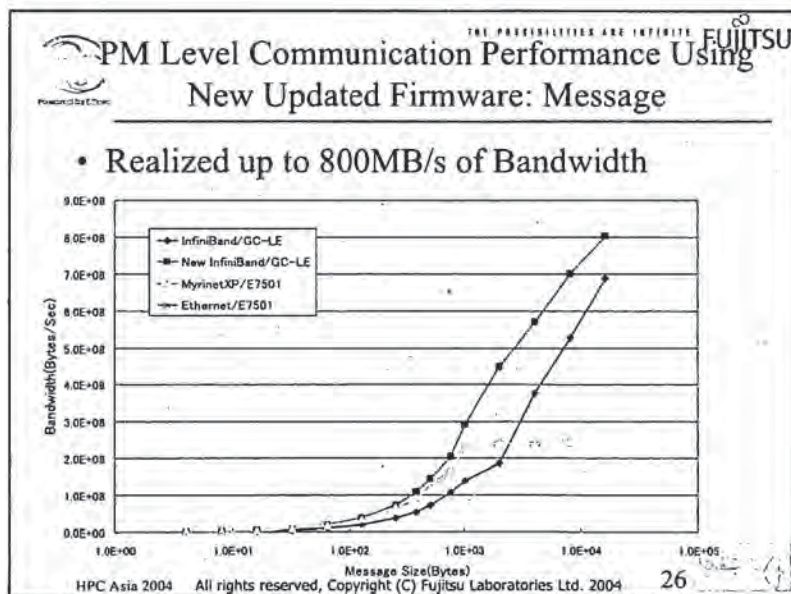
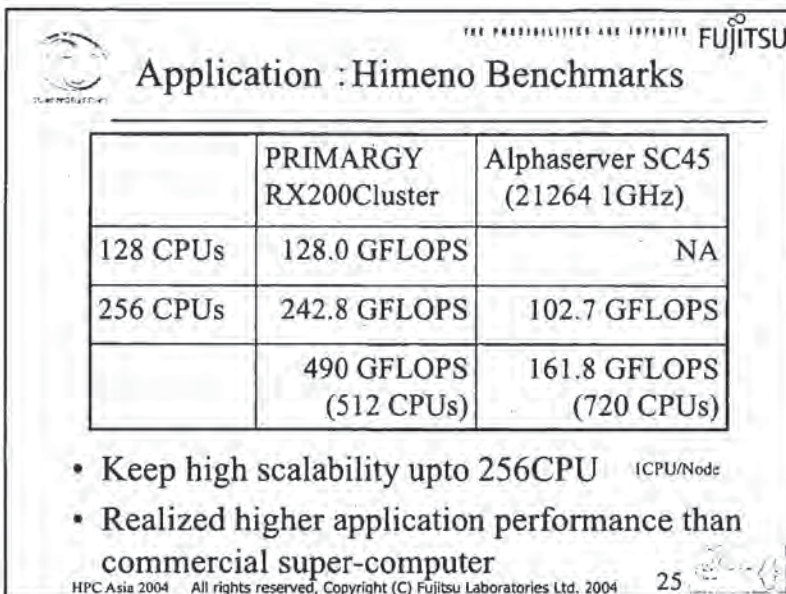
- PM/InfiniBand-FJ 688MB/s: 2.8 times Higher than that of MyrinetXP

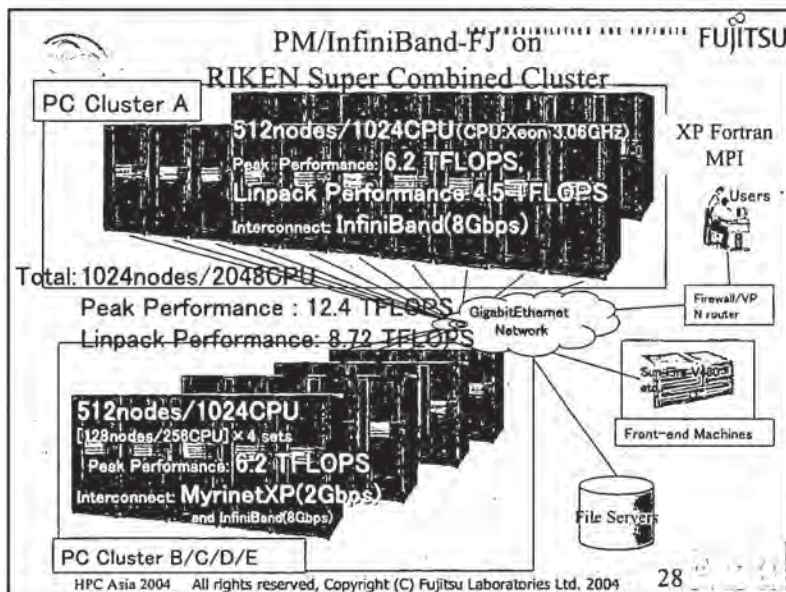
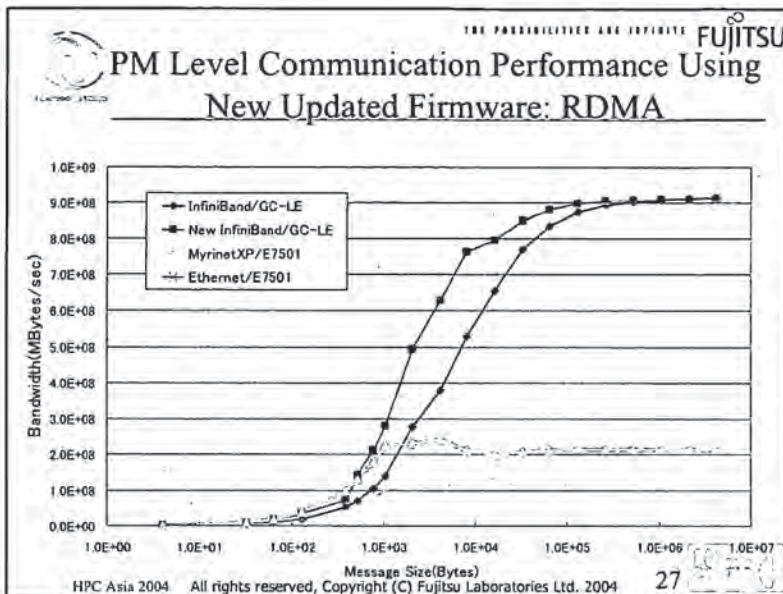


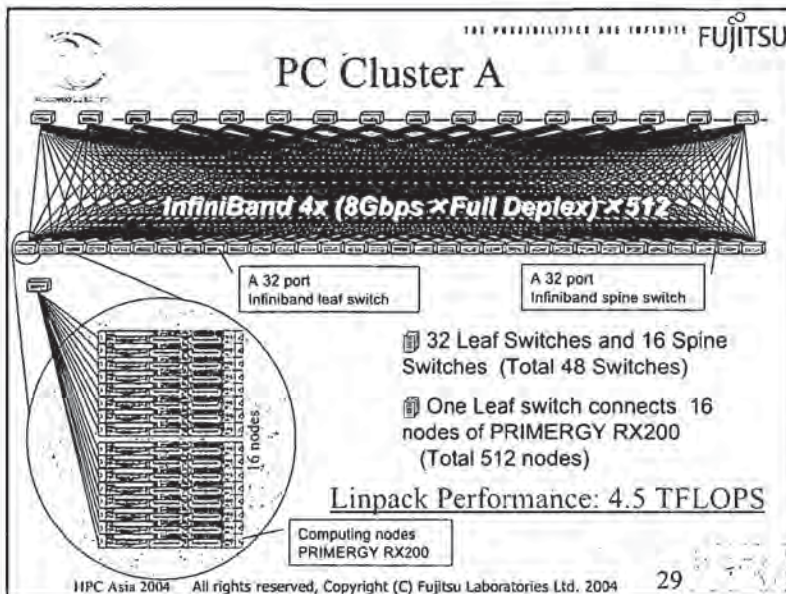
HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004

22









THE POSSIBILITIES ARE INFINITE **FUJITSU**

The RIKEN Cluster is #7 on TOP500

24th List (2004.June)

#	System name, Processor, Network	Rmax (GFlops)	NProc	Type
1	Earth Simulator	35,860	5,120	Vector
2	Thunder - Itanium2, 1.4 GHz	13,880	8,192	PC Cluster
3	ASCI Q - AlphaServer SC45, 1.25 GHz	10,280	2,200	Scalar
4	BlueGene LDD1 Prototype, PPC 440 0.5GHz	11,680	8,192	Scalar
5	Tungsten, Xeon 3.06 GHz, Myrinet	9,819	2,500	PC Cluster
6	eServer p690, Power4+ 1.9GHz	8,955	2,112	Scalar
7	RIKEN Super Combined Cluster, Xeon 3.06, (InfiniBand(FJ+Topspin)+MyrinetXP) GigaE	8,728	2,048	PC Cluster
8	BlueGene LDD2 Prototype, PPC 440 0.7GHz	8,655	4,096	Scalar
9	MPP2, Itanium2 1.5GHz, Quadrics	8,633	1,936	PC Cluster
10	Dawning4000A Opteron 2.2GHz, Myrinet	8,061	2,560	PC Cluster

HPC Asia 2004 All rights reserved, Copyright (C) Fujitsu Laboratories Ltd. 2004 30



Summary

- A Design of High Performance Communication Facility PM/InfiniBand-FJ for Large Scale PC Clusters
 - Realizing high communication performance and high reliability by extending the specifications of InfiniBand
 - Introducing ASQP-SROP to reduce receive buffer consumption
 - Reducing communication latency by eliminating number of PCI DMA
 - Realizing Checkpoint-restart function by using SCore
- High Communication Performance: 913.2MB/s (RDMA)
- High Application Performance:
 - NPB IS Class B: 1.52 times faster than that of Myrinet XP on 128CPU
- Several Customers use PM/InfiniBand-FJ
 - RIKEN, NAREGI



Questions?

SCore can be Downloaded form
<http://www.pccluster.org>

Infiniband Scalability in Open MPI

G. M. Shipman^{1,2}, T. S. Woodall¹, R. L. Graham¹, A. B. Maccabe²

¹ Advanced Computing Laboratory
Los Alamos National Laboratory

² Scalable Systems Laboratory
Computer Science Department
University of New Mexico

Abstract

Infiniband is becoming an important interconnect technology in high performance computing. Recent efforts in large scale Infiniband deployments are raising scalability questions in the HPC community. Open MPI, a new production grade implementation of the MPI standard, provides several mechanisms to enhance Infiniband scalability. Initial comparisons with MVAPICH, the most widely used Infiniband MPI implementation, show similar performance but with much better scalability characteristics. Specifically, small message latency is improved by up to 10% in medium/large jobs and memory usage per host is reduced by as much as 300%. In addition, Open MPI provides predictable latency that is close to optimal without sacrificing bandwidth performance.

1 Introduction

High performance computing (HPC) systems are continuing a trend toward distributed memory clusters consisting of commodity components. Many of these systems make use of commodity or 'near' commodity interconnects

including Myrinet [17], Quadrics [3], Gigabit Ethernet and, recently, Infiniband [1]. Infiniband (IB) is increasingly deployed in small to medium sized commodity clusters. It is IB's low price/performance qualities that has made it attractive to the HPC market.

Of the available distributed memory programming models, the Message Passing Interface (MPI) standard [16] is currently the most widely used. Several MPI implementations support Infiniband including Open MPI [10], MVAPICH [15], LA-MPI [11] and NCSA MPI [18]. However, there are concerns about the scalability of Infiniband for MPI applications, partially arising from the fact that Infiniband was initially developed as a general I/O fabric technology and not specifically targeted to HPC [4].

In this paper, we describe Open MPI's scalable support for Infiniband. In particular, Open MPI makes use of Infiniband feature not currently used by other MPI/IB implementations, allowing Open MPI to scale more effectively than current implementations. We illustrate the scalability of Open MPI's Infiniband support through comparisons with the widely-used MVAPICH implementation, and show that Open MPI uses less memory and provides better latency than MVAPICH on medium/large-scale

clusters.

The remainder of this paper is organized as follows. Section 2 presents a brief overview of the Open MPI general point-to-point message design. Next, section 3 discusses the Infiniband architecture including current limitations of the architecture. MVAPICH is discussed in section 4 including potential scalability issues relating to this implementation. Section 5 provides a detailed description of Infiniband support in Open MPI. Scalability and performance results are discussed in section 6, followed by conclusions and future work in section 7.

2 Open MPI

The Open MPI Project is a collaborative effort by Los Alamos National Lab, the Open Systems Laboratory at Indiana University, the Innovative Computing Laboratory at the University of Tennessee and the High Performance Computing Center at the University of Stuttgart (HLRS). The goal of this project is to develop a next generation implementation of the Message Passing Interface. Open MPI draws upon the unique expertise of each of these groups which includes prior work on LA-MPI, LAM/MPI [20], FT-MPI [9] and PAX-MPI [13]. Open MPI is however, a completely new MPI, designed from the ground up to address the demands of current and next generation architectures and interconnects.

Open MPI is based on a Modular Component Architecture [19]. This architecture supports the runtime selection of components that are optimized for a specific operating environment. Multiple network interconnects are supported through this MCA. Currently there are two Infiniband components in Open MPI. One supporting the OpenIB Verbs-API and another supporting the Mellanox Verbs-API. In addition to being highly optimized for scalability these components provide a number of performance

and scalability parameters which allow for easy tuning.

The Open MPI point-to-point (p2p) design and implementation is based on multiple MCA frameworks. These frameworks provide functional isolation with clearly defined interfaces. Figure 1 illustrates the p2p framework architecture.

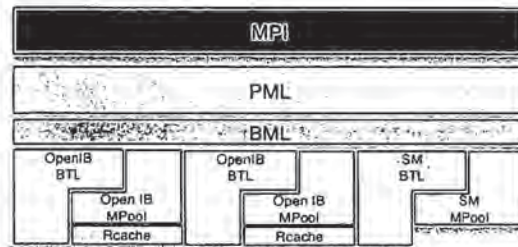


Figure 1: Open MPI p2p framework

As shown in Figure 1 the architecture consists of four layers. Working from the bottom up these layers are the Byte Transfer Layer (BTL), BTL Management Layer (BML), Point-to-Point Messaging Layer (PML) and the MPI layer. Each of these layers is implemented as an MCA framework. Other MCA frameworks shown are the Memory Pool (MPool) and the Registration Cache (Rcache). While these are illustrated and defined as layers, critical send/receive paths bypass the BML, as it is used primarily during initialization/BTL selection.

MPool The memory pool provides memory allocation/deallocation and registration/deregistration services. Infiniband requires memory to be registered (physical pages present and pinned) before send/receive or RDMA operations can use the memory as a source or target. Separating this functionality from other components allows the MPool to be shared

among various layers. For example, MPI_ALLOC_MEM uses these MPools to register memory with available interconnects.

Rcache The registration cache allows memory pools to cache registered memory for later operations. When initialized, MPI message buffers are registered with the Mpool and cached via the Rcache. For example, during an MPI_SEND the source buffer is registered with the memory pool and this registration may then be cached, depending on the protocol in use. During subsequent MPI_SEND operations the source buffer is checked against the Rcache, and if the registration exists the PML may RDMA the entire buffer in a single operation without incurring the high cost of registration.

BTL The BTL modules expose the underlying semantics of the network interconnect in a consistent form. BTLs expose a set of communication primitives appropriate for both send/receive and RDMA interfaces. The BTL is not aware of any MPI semantics; it simply moves a sequence of bytes (potentially non-contiguous) across the underlying transport. This simplicity will enable early adoption of novel network devices and encourages vendor support. There are several BTL modules currently available; including TCP, GM, Portals, Shared Memory (SM), Mellanox VAPI and OpenIB VAPI. In the later section we discuss the Mellanox VAPI and OpenIB VAPI BTLs.

BML The BML acts as a thin multi-plexing layer, allowing the BTLs to be shared among multiple upper layers. Discovery of peer resources is coordinated by the BML and cached for multiple consumers of the BTLs. After resource discovery, the BML

layer may be safely bypassed by upper layers for performance. The current BML component is named R2.

PML The PML implements all logic for p2p MPI semantics including standard, buffered, ready, and synchronous communication modes. MPI message transfers are scheduled by the PML based on a specific policy. This policy incorporates BTL specific attributes to schedule MPI messages. Short and long message protocols are implemented within the PML. All control messages (ACK/NACK/MATCH) are also managed at the PML. The benefit of this structure is a separation of transport protocol from the underlying interconnects. This significantly reduces both code complexity and code redundancy enhancing maintainability. There are currently three PMLs available in the Open MPI code base. This paper discusses OB1 the latest generation PML in the later section.

During startup, a PML component is selected and initialized. The PML component selected defaults to OB1 but may be overridden by a runtime parameter/environment setting. Next the BML component R2 is selected. R2 then opens and initializes all available BTL modules. During BTL module initialization, R2 directs peer resource discovery on a per-BTL basis. This allows the peers to negotiate which set of interfaces they will use to communicate with each other. This infrastructure allows for heterogeneous networking interconnects within a cluster.

3 Infiniband

The Infiniband specification is published by the Infiniband Trade Association (ITA) originally created by Compaq, Dell, Hewlett-Packard, IBM, Intel, Microsoft, and Sun Microsystems.

IB was originally proposed as a general I/O technology, allowing for a single I/O fabric to replace multiple existing fabrics. The goal of a single I/O fabric has faded and currently Infiniband is targeted as an Inter Process Communication (IPC) and Storage Area Network (SAN) interconnect technology.

Infiniband, similar to Myrinet and Quadrics, provides both Remote Direct Memory Access (RDMA) and Operating System (OS) bypass facilities. RDMA enables data transfer from the address space of an application process to a peer process across the network fabric without requiring involvement of the host CPU. Infiniband RDMA operations support both two-sided send/receive and one-sided put/get semantics. Each of these operations may be queued from the user level directly to the host channel adapter (HCA) for execution, bypassing the OS to minimize latency and processing requirements on the host CPU.

3.1 Infiniband OS Bypass

To enable OS bypass, Infiniband defines the concept of a Queue Pair (QP). The Queue Pair mechanism provides user level processes direct access to the IB HCA. Unlike traditional stack based protocols, there is no need to packetize the source buffer or process other protocol specific messages in the OS or at user level. Packetization and transport logic is located almost entirely in the HCA.

Each queue pair consists of both a send and receive work queue, and is additionally associated with a Completion Queue (CQ). Work Queue Entries (WQEs) are posted from the user level for processing by the HCA. Upon completion of a WQE, the HCA posts an entry to the completion queue, allowing the user level process to poll and/or wait on the completion queue for events related to the queue pair.

Two-sided send/receive operations are initiated by enqueueing a send WQE on a QP's send queue. The WQE specifies only the sender's local buffer. The remote process must pre-post a receive WQE on the corresponding receive queue which specifies a local buffer address to be used as the destination of the receive. Send completion indicates the send WQE is completed locally and results in a sender side CQ entry. When the transfer actually completes a CQ entry will be posted to the receiver's CQ.

One-sided RDMA operations are likewise initiated by enqueueing a RDMA WQE on the Send Queue. However, this WQE specifies both the source and target virtual addresses along with a protection key for the remote buffer. Both the protection key and remote buffer address must be obtained by the initiator of the RDMA read/write prior to submitting the WQE. Completion of the RDMA operation is local and results in a CQ entry at the initiator. The operation is one-sided in the sense that the remote application is not involved in the request and does not receive notification of its completion.

3.2 Infiniband Resource Allocation

Infiniband does place some additional constraints on these operations. As data is moved directly between the host channel adapter (HCA) and user level source/destination buffers, these buffers must be registered with the HCA in advance of their use. Registration is a relatively expensive operation which locks the memory pages associated with the request, thereby preserving the virtual to physical mappings. Additionally, when supporting send/receive semantics, pre-posted receive buffers are consumed in order as data arrives on the host channel adapter (HCA). Since no attempt is made to match available buffers to the incoming message size, the maximum size of a message is constrained to the minimum size of the posted receive buffers.

Infiniband additionally defines the concept of a Shared Receive Queue (SRQ). A single SRQ may be associated with multiple QPs during their creation. Receive WQEs that are posted to the SRQ are then shared resources to all associated QPs. This capability plays a significant role in improving the scalability of the connection-oriented transport protocols described below.

3.3 Infiniband Transport Modes

The Infiniband specification details five modes of transport

1. Reliable Connection (RC)
2. Reliable Datagram (RD)
3. Unreliable Connection (UC)
4. Unreliable Datagram (UD)
5. Raw Datagram

Reliable Connection provides a connection oriented transport between two queue pairs. During initialization of each QP, peers exchange addressing information used to bind the QP's and bring them to a connected state. Work requests posted on each QP's Send Queue are implicitly addressed to the remote peer. As with any connection oriented protocol, scalability may be a concern as the number of connected peers grows large, and resources are allocated to each QP. Both Open MPI and MVAPICH currently use RC transport modes.

Reliable Datagram allows a single QP to be used to send and receive messages to/from other RD QPs. Whereas in RC reliability state is associated with the QP, RD associates this state with an end-to-end (EE) context. The intent of the Infiniband specification is that the EE's will scale much more effectively with the number of active peers. Both Reliable Connection and Reliable

Datagram provide acknowledgment and retransmission. In practice, this portion of the specification has yet to be implemented.

Unreliable Connection and Unreliable Datagram are similar to their reliable counterparts in terms of QP resources. These transports differ in that they are unacknowledged services and do not provide for retransmission of dropped packets. The high cost of user reliability relative to the hardware reliability of RC and RD make these modes of transport inefficient for MPI.

3.4 Infiniband Summary

Infiniband shares many of the architectural features of VIA. Scalability limitations of VIA are well known [5] to the HPC community. These limitations arise from a connection oriented protocol, RDMA semantics and the lack of direct support for asynchronous progress. While the Infiniband specification does address scalability of connection oriented protocols through the RD transport mode, the industry leader Mellanox has yet to implement this portion of the specification. Additionally, while the SRQ mechanism addresses scalability issues associated with the reliable connection oriented transport, issues related to flow control and resource management must be considered. MPI implementations must therefore compensate for these limitations in order to effectively scale to large clusters.

4 MVAPICH

MVAPICH is currently the most widely used MPI implementation on Infiniband platforms. A descendent of MPICH [12], one of the earliest MPI implementation, as well as MVICH [14], MVAPICH provides several novel features for Infiniband support. These features include small message RDMA, caching of registered memory regions and multi-rail IB support.

4.1 Small Message Transfer

The MVAPICH design incorporates a novel approach to small message transfer. Each peer is pre-allocated and registered a separate memory region for small message RDMA operations called a persistent buffer association. Each of these memory regions is structured as a circular buffer allowing the remote peer to RDMA directly into the currently available descriptor. Remote completion is detected by the peer polling the current descriptor in the persistent buffer association. A single bit can indicate completion of the RDMA as current Mellanox hardware guarantees the last byte of an RDMA operation will be the last byte delivered to the application. This design takes advantage of the extremely low latencies of Infiniband RDMA operations.

Unfortunately, this is not a scalable solution for small message transfer. As each peer requires a separate persistent buffer, memory usage grows linearly with the number of peers. Polling each persistent buffer for completion also presents scalability problems. As the number of peers increases the additional overhead required to poll these buffers quickly erodes the benefits of small message RDMA.

A similar design was attempted earlier on ASCI Blue Mountain with what later evolved into LA-MPI to support HIPPI-800. The approach was later abandoned due to poor scalability and a hybrid approach evolved, taking advantage of HIPPI-800 firmware for multiplexing. Other alternative approaches to polling persistent buffers for completion have also been discussed and may prove to be more scalable [6].

To address the issues of small message RDMA, MVAPICH provides a medium and large configuration option. These options limit the resources used for small message RDMA and revert instead to standard send/receive. As demonstrated in our results section this configuration option improves the scalability of small

message latencies but still results in sub-optimal performance as the number of peers increases.

4.2 Connection Management

MVAPICH uses static connection management, establishing a fully connected job at startup. In addition to eagerly establishing QP connections, MVAPICH also allocates a persistent buffer association for each peer. If send/receive is used instead of small message RDMA, MVAPICH allocates receive descriptors on a per QP basis instead of using the shared receive queue across QP's. This further increases resource allocation per peer.

4.3 Caching Registered Buffers

As discussed earlier Infiniband requires all memory to be registered (pinned) with the HCA. Memory registration is an expensive operation so MVAPICH caches memory registrations for later use. This allows subsequent message transfers to queue a single RDMA operation without paying any registration costs. This approach to registration assumes that the application will reuse buffers often in order to amortize the high cost of a single up front memory registration. For some applications this is a reasonable assumption.

A potential issue when caching memory registrations is that the application may free a cached memory region and then return the associated pages to the OS¹. The application could later allocate another memory region and obtain the same virtual address as the previously freed buffer. Subsequent RDMA operations may use the cached registration but this registration may now contain incorrect virtual to physical mappings. RDMA operation may therefore use an

¹Memory is returned via the `sbrk` function in UNIX and Linux.

unintentional memory region. In order to avoid this scenario MVAPICH forces the application to never release pages to the OS² and thereby preserving virtual to physical mappings. This approach may cause resource exhaustion as the OS can never reclaim physical pages.

5 Design of Open MPI

In this section we discuss Open MPI's support for Infiniband, including techniques to enhance scalability.

5.1 The OB1 PML Component

OB1 is the latest point-to-point management layer for Open MPI. OB1 replaces the previous generation PML - TEG [21]. The motivation for a new PML was driven by code complexity at the lower layers. Previously much of the MPI p2p semantics such as the short and long protocols were duplicated for each interconnect. This logic as well as RDMA specific protocol logic was moved up to the PML layer. Initially there was concern that moving this functionality into an upper layer would cause performance degradation. Preliminary performance benchmarks have shown this not to be the case. This restructuring has substantially decreased code complexity while maintaining performance on par with both previous Open MPI architectures as well as other MPI implementations. Through the use of device appropriate abstractions we have exposed the underlying architecture to the PML level. As such the overhead of the p2p architecture in Open MPI is lower than that of other MPI implementations.

OB1 provides numerous features to support both send/receive and RDMA read/write operations. The send/receive protocol uses pre-

allocated/registered buffers to copy in for send and copy out for receive. This protocol provides good performance for small messages transfer and is used both for the eager protocol as well as control messages.

To support RDMA operations, OB1 makes use of the Mpool and Rcache components in order to cache memory regions for later RDMA operations. Both source and target buffers must be registered prior to an RDMA read or write of the buffer. Subsequent RDMA operation can make use of pre-registered memory in the Mpool/Rcache. While MVAPICH prevents physical pages from being released to the OS, Open MPI instead uses memory hooks to intercept deallocation of memory. When memory is deallocated it is checked against the Rcache and all matching registrations are de-registered. This prevents future use of an invalid memory registration while allowing memory to be returned to the host operating system.

In addition to supporting both send/receive and RDMA read/write, Open MPI provides a hybrid RDMA pipeline protocol. This protocol avoids caching of memory registrations and virtually eliminates memory copies. The protocol begins by eagerly sending data using send/receive up to a configurable eager limit. Upon receipt and match the receiver responds with an ack to the source and begins registering blocks of the target buffer across the available HCAs. The number of blocks registered at any given time is bound by a configurable pipeline depth. As each registration in the pipeline completes an RDMA control message is sent to the source to initiate an RDMA write on the block.

To cover the cost of initializing the pipeline, on receipt of the initial ack at the source, send/receive semantics are used to deliver data from the eager limit up to the initial RDMA write offset. As RDMA control messages are received at the source, the corresponding block

²The `malloc` function in UNIX and Linux prevents pages from being given back the OS.

of the source buffer is registered and an RDMA write operation initiated on the current block. On local completion at the source, an RDMA FIN message is sent to the peer. Registered blocks are de-registered upon local completion or receipt of the RDMA FIN message. If required, the receipt of an RDMA FIN messages may also further advance the RDMA pipeline.

This protocol effectively overlaps the cost of registration/deregistration with RDMA writes. Resources are released immediately and the high overhead of a single large memory registration is avoided. Additionally, this protocol results in improved performance for applications that seldom reuse buffers for MPI operations.

5.2 The OpenIB and Mvapi BTLs

This section focuses on two BTL components, both of which support the Infiniband interconnect. These two components are called Mvapi, based on the Mellanox verbs API, and OpenIB, based on the OpenIB verbs API. Other than this difference the Mvapi and OpenIB BTL components are nearly identical. Two major goals drove the design and implementation of these BTL components, performance and scalability. The following details the scalability issues addressed in these components.

5.2.1 Connection Management

As detailed earlier, connection oriented protocols pose scaling challenges for larger clusters. In contrast to the static connection management strategy adopted by MVAPICH, Open MPI uses dynamic connection management. When one peer first initiates communication with another peer, the request is queued at the BTL layer. The BTL then establishes the connection through an out of band (OOB) channel. After connection establishment, queued sends are progressed to the peer. This results in a shorter startup time

and a longer first message latency time for Infiniband communication. Resource usage reflects the actual communication patterns of the application and not the number of peers in the MPI job. As such, MPI codes with scalable communication patterns will require fewer resources.

5.2.2 Small Message Transfer

MVAPICH uses a pure RDMA protocol for small message transfer requiring a separate buffer per peer. Open MPI currently avoids this scalability problem by using Infiniband's send/receive interface for small messages. In an MPI job with 64 nodes, instead of polling 64 preallocated memory regions for remote RDMA completion, Open MPI polls a single completion queue. Instead of preallocating 64 separate memory regions for RDMA operations, Open MPI will optionally post receive descriptors to the SRQ. Unfortunately, Infiniband does not support flow control when the SRQ is used. As such Open MPI provides a simple user level flow control mechanism. As demonstrated in our results, this mechanism is probabilistic and may result in retransmission under certain communication patterns and may require further analysis.

Open MPI's resource allocation scheme is detailed in the Figure 2. Per peer resources include 2 Reliable Connection QP's, one for High Priority transfers and one for Low Priority transfers. High priority QP's share a single Shared Receive Queue and Completion Queue as do low priority QP's. Receive descriptors are posted to the SRQ on demand. The number of receive descriptors posted to the SRQ is calculated using the following method:

$$x = \log_2(n) * k + b$$

- x Number of Receive Descriptors to post
- n Number of peers in cluster
- k per peer scaling factor for number of Receive Descriptors to post
- b base number of Receive Descriptors to post

The high priority QP is for small control messages and any data sent eagerly to the peer. The low priority QP is for larger MPI level messages as well as all RDMA operations. Using two QP's allows Open MPI to maintain two sizes of receive descriptors, an eager size for the high priority QP and a maximum send size for the low priority QP. While requiring an additional QP per peer, we gain a finer grained control over receive descriptor memory usage. In addition, using two QPs allows us to exploit parallelism available in the HCA hardware [8].

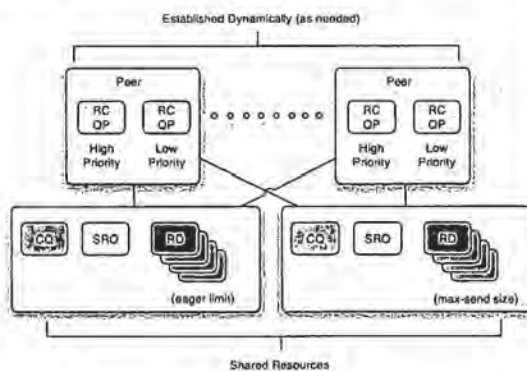


Figure 2: Open MPI Resource Allocation

5.3 Asynchronous progress

A further problem in RDMA devices is lack of direct support for asynchronous progress. Asynchronous progress in MPI is the ability for the MPI library to make progress on both sending and receiving of messages when the application has left the MPI library. This allows for effective overlap of communication and computation.

RDMA based protocols require that the initiator of the RDMA operation be aware of both the source and destination buffers. To avoid a memory copy and to allow the user to send and receive from arbitrary buffers of arbitrary length the peer's memory region must be obtained by the initiator prior to each request.

Figure 3 illustrates a timing of a typical RDMA transfer in MPI using an RDMA Write. The RTS, CTS and FIN can either be sent using send/receive or small message RDMA. Either method requires the receiver to be in the MPI library to progress the RTS, send the CTS and then to handle the completion of the RDMA operation by receiving the FIN message.

In contrast to traditional RDMA interfaces, one method of providing asynchronous progress is by moving the matching of the receive buffer of the MPI message to the network interface. Portals [7] style interfaces allow this by associating target memory locations with the tuple of MPI communicator, tag, and sender address thereby eliminating the need for the sender to obtain the receivers target memory address.

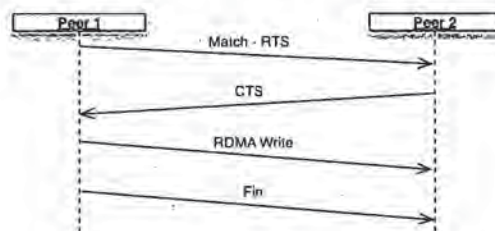


Figure 3: RDMA Write

From Figure 3 we can see that if the receiver is not currently in the MPI library on the initial RDMA of the RTS, no progress is made on the RDMA write until after the receiver enters the MPI library.

Open MPI addresses asynchronous progress for Infiniband by introducing a progress thread.

The progress thread allows the Open MPI library to continue to progress messages by processing RTS/CTS and FIN messages. While this is a solution to asynchronous progress, the cost in terms of message latency is quite high. In spite of this, some applications may benefit from asynchronous progress even in the presence of higher message latency. This is especially true if the application is written in a manner to take advantage of communication/computation overlap.

6 Results

This section presents a comparison of our work. First we present scalability results in terms of per node resource allocation. Next we examine performance results, showing that while Open MPI is highly scalable it also provides excellent performance in the NAS Parallel benchmark (NPB) [2].

6.1 Scalability

As demonstrated earlier, the memory footprint of a pure RDMA protocol as used in MVAPICH increases linearly with the number of peers. This is partially due to lack of dynamic connection management as well as resource allocation. Resource allocation for the small RDMA protocol is per peer. Specifically, each peer is allocated a memory region in every other peer. As the number of peers increases this memory allocation scheme becomes intractable. Open MPI avoids these costs in two ways. First, Open MPI establishes connections dynamically on the first send to a peer. This allows resource allocation to reflect the communication pattern of the MPI application. Second, Open MPI optionally makes use of the Infiniband SRQ so that receive resources (pre-registered memory) can be shared among multiple endpoints.

To examine memory usage of the MPI library we have used three different benchmarks. The first is a simple “hello world” application that does not communicate with any of its peers. This benchmark establishes a baseline of memory usage for an application. Figure 4 demonstrates that Open MPI’s memory usage is constant as no connections are established and therefore no resources are allocated for other peers. MVAPICH on the other hand preallocates resources for each peer at startup so memory resources increase as the number of peers increases. Both MVAPICH small and medium configurations consume more resources.

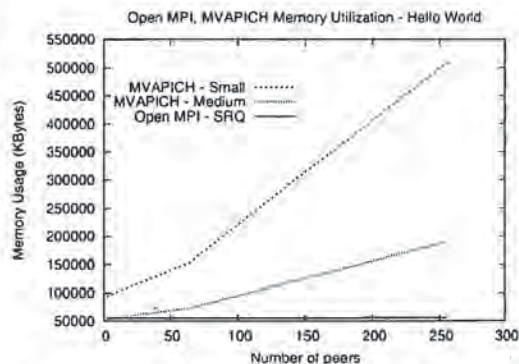


Figure 4: Hello World Memory Usage

Our next benchmark is a pairwise ping-pong, where peer’s of neighbor rank ping each other, that is rank 0 pings rank 1 and rank 2 pings rank 3 and so on. As Figure 5 demonstrates, Open MPI memory usage is constant. This is due to dynamic connection management, only peers participating in communication are allocated resources. Again we see that MVAPICH memory usage ramps up with the number of peers.

Our final memory usage benchmark is a worst case for Open MPI, each peer communicates with every other peer. As can be seen in Figure 6 Open MPI SRQ memory usage does increase as the number of peers increases, but at a

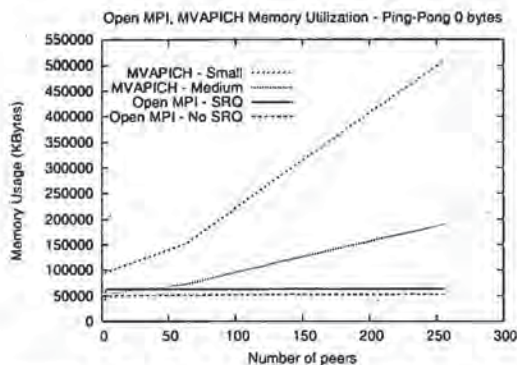


Figure 5: Pairwise Ping-Pong Memory Usage

much smaller rate than that of MVAPICH. This is due to the use of the SRQ for resource allocation. Open MPI without SRQ scales slightly worse than the MVAPICH medium configuration, this is due to Open MPI's use of two QPs per peer.

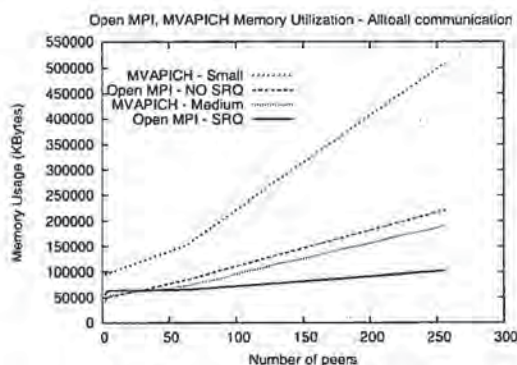


Figure 6: All-to-all Memory Usage

6.2 Performance

To verify the performance of our MPI implementation we present both micro benchmarks as well as the NAS Parallel Benchmarks

6.2.1 Latency

Ping-pong latency is a standard benchmark of MPI libraries. As with any micro-benchmark, ping-pong provides only part of the true representation of performance. Most ping-pong results are presented using two nodes involved in communication. While this number provides a lower bound on communication latency, multi-node ping-pong is more representative of communication patterns in anything but trivial applications. As such, we present ping-pong latencies for a varying number of nodes in which N nodes perform the previously discussed pairwise ping-pong. This enhancement to the ping-pong benchmark helps to demonstrate scalability of small message transfers because in larger MPI jobs the number of peers communicating at the same time often increases.

In this test, the latency of a zero byte message is measured for each pair of peers. We have then plotted the average with errorbars for each of these runs. As can be seen in Figure 7, the small message RDMA mechanisms provided in MVAPICH provides a benefit with a small number of peers. Unfortunately, the polling of memory regions is not a scalable architecture as can be seen when the number of peers participating in the latency benchmark increases. For each additional peer involved in the benchmark, every other peer must allocate and poll an additional memory region. Costs of polling quickly erode any improvements in latency. Memory usage is also higher on a per peer and aggregate basis. This trend occurs in both small and medium MVAPICH configurations. Open MPI provides much more predictable latencies and outperforms MVAPICH latencies as the number of peers increases. Open MPI - SRQ latencies are a bit higher than Open MPI - No SRQ latencies as the SRQ path under Mellanox HCA's is more costly.

The following Table 1 shows the Open MPI

send/receive latencies trail MVAPICH small message RDMA latencies but are better than MVAPICH send/receive latencies. This is an important result as larger MVAPICH clusters will make more use of send/receive and not small message RDMA.

	Average Latency
Open MPI - Optimized	5.64
Open MPI - Default	5.94
MVAPICH - RDMA	4.19
MVAPICH - Send/Receive	6.51

Table 1: Two node Ping-Pong latency in μ -sec. Optimized - Limits the number of WQE on the RQ Defaults - Default number of WQE on the RQ

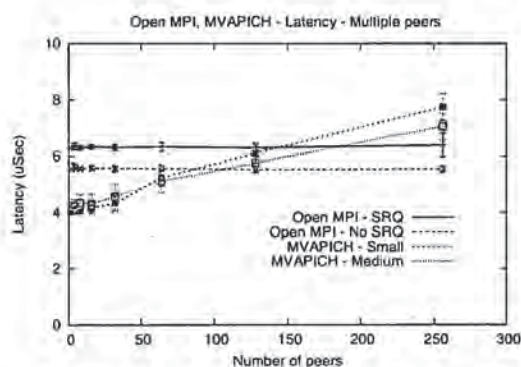


Figure 7: Multi-Node Zero Byte Latency

6.2.2 NPB

To demonstrate the performance of our implementation outside of micro benchmarks we used the NAS Parallel Benchmarks [2]. NPB is a set of benchmarks derived from computational fluid dynamics applications. All NPB benchmarks were run using the class C size of problem and all results are given in run-time (seconds). The

results of these benchmarks are summarized in Table 2. Open MPI was run using 3 configurations, with SRQ, SRQ with simple flow control and without SRQ. MVAPICH was run in both small and medium cluster configurations. Open MPI without SRQ and MVAPICH performance is similar. With SRQ, Open MPI performance is similar for the BT, CG, and EP benchmarks. BT, FT and IS performance is lower with SRQ as receive resources are quickly consumed in collective operations. Our current flow control mechanism addresses this issue for the BT benchmark but both the FT and IS benchmarks are still effected due to global broadcast and all-to-all communication patterns respectively. Further research into SRQ flow control techniques are ongoing.

6.3 Experimental Setup

Our experiments were performed on two different machine configurations. Two node ping-pong benchmarks were performed on dual Intel Xeon X86-64 3.2 Ghz processors with 2GB of RAM, and Mellanox PCI-Express Lion-Cub adapters connected via a Voltair 9288 switch. The Operating System is Linux 2.6.13.2 with Open MPI pre-release 1.0 and MVAPICH 0.9.5-118. All other benchmarks were performed on a 256 node cluster consisting of dual Intel Xeon X86-64 3.4 Ghz processors with a minimum 6GB of RAM, Mellanox PCI-Express Lion Cub adapters also connected via a Voltair switch. The Operating System is Linux 2.6.9-11 with Open MPI pre-release 1.0 and MVAPICH 0.9.5-118.

7 Future Work - Conclusions

Open MPI addresses many of the concerns regarding the scalability and use of Infiniband in HPC. In this section we summarize the results

	BT		CG				EP			
Nodes	64	256	32	64	128	256	32	64	128	256
Open MPI - No SRQ	100.03	25.03	20.17	12.74	7.39	5.56	38.89	19.84	9.95	5.11
Open MPI - SRQ	114.92	26.92	20.45	12.86	7.49	5.61	38.85	19.72	10.04	5.26
Open MPI - SRQ FC	100.13	25.33	21.13	12.83	7.38	5.63	39.10	19.76	12.88	5.12
MVAPICH - Small	98.78	27.40	20.33	12.96	7.84	6.11	39.15	19.65	10.02	5.32
MVAPICH - Large	99.22	27.58	20.24	13.15	7.83	6.09	39.10	19.59	9.89	5.31

	SP		FT				IS			
Nodes	64	256	32	64	128	256	32	64	128	256
Open MPI - No SRQ	54.39	16.08	36.64	18.28	9.39	4.81	2.23	1.62	0.97	0.52
Open MPI - SRQ	140.81	22.53	75.48	68.36	56.92	26.96	32.21	33.29	25.06	21.97
Open MPI - SRQ FC	54.90	14.61	54.81	35.87	19.39	24.54	5.32	4.38	12.35	11.12
MVAPICH - Small	53.66	15.16	37.59	19.42	10.17	4.84	2.19	1.55	0.87	0.42
MVAPICH - Large	53.87	15.84	37.91	19.51	9.85	4.88	2.20	1.56	0.87	0.50

Table 2: NPB Results - Each benchmark uses the class C option with a varying number of nodes, 1 process per node. Results are given in seconds.

of this paper and provide directions for future work. to a much higher level.

7.1 Conclusions

Open MPI's Infiniband support provides several techniques to improve scalability. Dynamic connection management allows per peer resource usage to reflect the applications chosen communication pattern, thereby allowing scalable MPI codes to preserve resources. Per peer memory usage in these types of applications will be significantly less in Open MPI when compared to other MPI implementations which lack this feature. Optional support for an asynchronous progress thread addresses the lack of direct support for asynchronous progress within Infiniband, potentially further reducing buffering requirements at the HCA. Shared resource allocation scales much more effectively than per peer resource allocation through the use of the Infiniband Shared Receive Queue (SRQ). This should allow even fully connected applications to scale

7.2 Future work

This work has identified additional areas for improvement. As the NAS parallel benchmarks illustrated, there are concerns regarding the SRQ case that require further consideration. Preliminary results indicate that an effective flow control and/or resource replacement policy must be implemented, as resource exhaustion results in significant performance degradation.

Additionally, Open MPI currently utilizes an OOB communication channel for connection establishment, which is based on TCP/IP. Using an OOB channel based on the unreliable datagram protocol will decrease first message latency and potentially improve the performance of the Open MPI run-time environment.

While connections are established dynamically, once opened, all connections are persistent. Some MPI codes which randomly communicate with peers may experience high resource

usage even if communication with the peer is infrequent. For these types of applications, dynamic connection teardown may be beneficial.

Open MPI currently supports both caching of RDMA registrations as well as a hybrid RDMA pipeline protocol. The RDMA pipeline provides good results even in applications that rarely reuse application buffers. Currently Open MPI does not cache RDMA registrations used in the RDMA pipeline protocol. Caching these registration would allow subsequent RDMA operations to avoid the cost of registration/deregistration if the send/rcv buffer is used more than once, while providing good performance even when the buffer is not used again.

Acknowledgments

The authors would like to thank Kurt Ferreira and Patrick Bridges of UNM and Jeff Squyres and Brian Barrett of IU for comments and feedback on early versions of this paper.

References

- [1] Infiniband Trade Association. Infiniband architecture specification vol 1, release 1.2, 2004.
- [2] Bailey, Barszcz, Barton, Browning, Carter, Dagum, Fatoohi, Fineberg, Frederickson, Lasinski, Schreiber, Simon, Venkatakrishnan, and Weeratunga. NAS parallel benchmarks, 1994.
- [3] Jon Beecroft, David Addison, Fabrizio Petrini, and Moray McLaren. QsNetII: An interconnect for supercomputing applications, 2003.
- [4] R. Brightwell, D. Doerfler, and K.D. Underwood. A comparison of 4x infiniband and quadrics elan-4 technologies. In *Proceedings of 2004 IEEE International Conference on Cluster Computing*, pages 193–204, September 2004.
- [5] R. Brightwell and A. Maccabe. Scalability limitations of VIA-based technologies in supporting MPI. In *Proceedings of the Fourth MPI Developer's and User's Conference*, March 2000.
- [6] Ron Brightwell. A new MPI implementation for cray SHMEM. In *PVM/MPI*, pages 122–130, 2004.
- [7] Ron Brightwell, Tramm Hudson, Arthur B. Maccabe, and Rolf Riesen. The portals 3.0 message passing interface, November 1999.
- [8] V Velusamy et al. Programming the infiniband network architecture for high performance message passing systems. In *Proceedings of The 16th IASTED International Conference on Parallel and Distributed Computing and Systems*, 2004.
- [9] G. E. Fagg, A. Bukovsky, and J. J. Dongarra. HARNESS and fault tolerant MPI. *Parallel Computing*, 27:1479–1496, 2001.
- [10] E. Garbriel, G.E. Fagg, G. Bosilica, T. Angskun, J. J. Dongarra J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R.H. Castain, D.J. Daniel, R.L. Graham, and T.S. Woodall. Open MPI: goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 2004.
- [11] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. D. Risinger, and M. W. Sukalski. A network-failure-tolerant

- message-passing system for terascale clusters. *International Journal of Parallel Programming*, 31(4), August 2003.
- [12] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.
 - [13] Rainer Keller, Edgar Gabriel, Bettina Krammer, Matthias S. Mueller, and Michael M. Resch. Towards efficient execution of MPI applications on the grid: porting and optimization issues. *Journal of Grid Computing*, 1:133–149, 2003.
 - [14] Lawrence Berkeley National Laboratory. Mvich: Mpi for virtual interface architecture, August 2001.
 - [15] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High performance RDMA-based MPI implementation over infiniband. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 295–304, New York, NY, USA, 2003. ACM Press.
 - [16] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, November 1993.
 - [17] Myricom. Myrinet-on-VME protocol specification.
 - [18] S. Pakin and A. Pant. . In *Proceedings of The 8th International Symposium on High Performance Computer Architecture (HPCA-8)*, Cambridge, MA, February 2002.
 - [19] Jeffrey M. Squyres and Andrew Lumsdaine. The component architecture of open MPI: Enabling third-party collective algorithms. In Vladimir Getov and Thilo Kielmann, editors, *Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, pages 167–185, St. Malo, France, July 2004. Springer.
 - [20] J.M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, Venice, Italy, September / October 2003. Springer-Verlag.
 - [21] T.S. Woodall, R.L. Graham, R.H. Castain, D.J. Daniel, M.W. Sukalsi, G.E. Fagg, E. Garbriel, G. Bosilica, T. Angskun, J. J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine. Open MPI's TEG point-to-point communications methodology : Comparison to existing implementations. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 2004.

Software Architecture Specification (SAS)

Revision – 1.0.1

Last Print Date: 8/1/2002 - 1:48 PM

Linux System Software for the InfiniBand* Architecture

*Other names and brands may be claimed as the property of others.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Intel disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

This Specification as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

***Other names and brands may be claimed as the property of others.**

Copyright © 2002 Intel Corporation.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

Abstract

InfiniBand is an industry standard that defines a new input-output subsystem designed to connect processor nodes and I/O nodes to form a system area network. The specification primarily defines the hardware electrical, mechanical, link-level, and management aspects of an InfiniBand™ fabric, but does not define the lowest layers of the operating system stack needed to communicate over an InfiniBand fabric. The remainder of the operating system stack to support storage, networking, IPC, and systems management is left to the operating system vendor for definition.

This specification details a software architecture for the operating system software components needed to support an InfiniBand fabric, specifically for the Linux operating system. The architecture for several of these components is further influenced by other standards and emerging standards that define standards and protocols for components of the operating system. Examples here are emerging protocols like Internet protocol (IP) over InfiniBand (IPoIB) and the SCSI RDMA Protocol (SRP) and proposed definitions for standard InfiniBand transport and driver APIs. Taking into account the InfiniBand specification and these emerging standards, this specification details the overall architecture for the Linux operating system components.

This architecture serves as an overall umbrella for the development of these Linux operating system components by the Linux community. The goal is to have InfiniBand support completed and included in a standard Linux distribution in time for HCA product launch.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

Table of Contents

Table of Contents	iv
1. Introduction	8
1.1 Purpose of this Document	8
1.2 Document Scope	8
1.3 Terminology	9
1.3.1 Acronyms	9
1.4 Related Documents	10
2. Architecture Overview	12
2.1 Identification	12
2.2 Goals and Objectives	12
2.2.1 Compatibility and Interoperability	12
2.2.2 Extensibility	12
2.2.3 Portability and Platform Independence	12
2.3 The Architectural Model	13
2.3.1 Architectural Concepts and Representations	13
2.4 Software Architectural Overview	14
2.4.1 Architecture Block Diagram	14
2.4.2 Architecture Block Diagram Description	14
3. Architectural Specification – HCA Verbs Provider Driver	17
3.1 Introduction	17
3.2 System Structural Overview	17
3.2.1 Software Component View	17
3.2.2 Relationship between InfiniBand Access and HCA Components	18
3.3 Theory of Operation	22
3.3.1 Driver Interfaces	22
3.3.1.1 Driver initialization Interfaces	22
3.3.1.2 Driver close, unload, and cleanup	23
3.3.2 Typical Driver Usage Model	23
4. Architectural Specification – Access Layer	28
4.1 Introduction	28
4.2 Design Methodology	28
4.2.1 Functional Services	28
4.3 Theory of Operation	29
4.3.1 Management Services	29
4.3.1.1 Management Helper Functions	31
4.3.1.2 Communication Manager	36
4.3.1.3 Device Management Agent	37
4.3.1.4 Plug and Play Manager	37
4.3.2 Resource Management	39
4.3.3 Work Processing	40
4.3.3.1 General Work Processing	40
4.3.3.2 Unreliable Datagram Services	41

Rev. 1.0.1 - 01/Aug/02 01:48 PM

4.3.3.3	MAD Work Processing	41
4.3.4	Memory Manager.....	42
4.3.5	Operation Examples.....	43
4.3.5.1	Establishing a Connection to an IOC.....	43
4.3.5.2	Client Generated MAD Transaction	44
4.3.5.3	Joining a Multicast Group.....	45
5.	Architectural Specification – Subnet Management	46
5.1	Introduction	46
5.2	System Overview.....	46
5.2.1	Relevant Specifications.....	48
5.2.2	List of External Interfaces.....	48
5.2.2.1	APIs.....	48
5.2.3	List of Product Deliverables	48
5.2.3.1	User Interfaces.....	48
5.2.3.2	Product Documentation.....	48
5.3	Theory of Operation	48
5.3.1	Initialization	48
5.3.2	SM Functionality and Interfaces.....	48
5.3.3	SA Functionality	49
5.3.3.1	SA Query Data Path.....	49
5.3.4	OpenSM Internals in Brief.....	49
5.3.5	OpenSM Multi-threaded Discovery.....	50
5.3.6	OpenSM Non-disruptive Subnet Configuration	50
5.3.7	LMC > 0.....	50
6.	Architectural Specification – SRP Driver.....	51
6.1	Introduction	51
6.2	Overview	51
6.3	Theory of Operation	54
6.3.1	Load / Unload	54
6.3.2	Initialization.....	55
6.3.3	Connection Management	56
6.3.3.1	Connection Establishment.....	56
6.3.3.2	Connection Failover.....	57
6.3.4	Command Processing.....	57
6.3.5	Asynchronous IO	58
6.3.6	Transaction Ordering	58
6.3.7	Multithreading.....	58
6.3.8	Interrupt Processing	58
6.3.9	Error Handling	58
6.3.9.1	Transport Errors	59
6.3.9.2	Invalid Requests	59
6.3.9.3	Device Errors.....	59
6.3.10	OS Interfaces	59
6.3.11	InfiniBand Access Interfaces.....	59
7.	Architectural Specification – IPoIB Driver	60
7.1	Introduction	60
7.2	System Structural Overview.....	60
7.2.1	Pictorial of Components Structure.....	61
7.2.2	Component Details.....	62

Rev. 1.0.1 - 01/Aug/02 01:48 PM

External interfaces.....	62
Internal interfaces.....	62
7.2.3 List of Third Party Components	62
7.2.4 List of External Interfaces.....	63
7.2.4.1 OS Interfaces	63
7.2.4.2 APIs.....	63
7.2.5 List of Product Deliverables	63
7.2.5.1 User Interfaces.....	63
7.2.5.2 Programs without Interfaces.....	63
7.2.5.3 Programmatic Interfaces (APIs).....	63
7.2.5.4 Internal Libraries.....	63
7.2.5.5 Product Documentation.....	64
7.3 Theory of Operation	64
7.3.1 Network Adapter Creation and Management	64
7.3.2 Mapping IBA Address Vectors to Adapter Hardware Addresses	64
7.3.3 ARP and Neighbor Discovery	65
7.3.4 QP Usage	66
8. Architectural Specification – Offload Sockets Framework and Sockets Direct Protocol (SDP).....	67
8.1 Introduction	67
8.2 Requirements for Offload Sockets Framework in Linux	68
8.3 System Structural Overview.....	70
8.3.1 Existing Sockets Architecture in Linux	70
8.3.2 Limitations in Existing Linux Sockets Architecture.....	71
8.3.3 Evaluations of Alternative Architectures.....	72
8.3.4 Software Component View.....	73
8.3.5 Component Details:	75
8.3.5.1 Offload Protocol Switch.....	75
External interfaces.....	75
Internal interfaces.....	75
8.3.5.2 Offload Protocol Module - Sockets Direct.....	76
External interfaces.....	76
Internal interfaces.....	76
8.3.5.3 Offload Transport Module - InfiniBand Transport	77
External interfaces.....	77
Internal interfaces.....	77
8.3.6 Product Deliverables.....	78
8.3.6.1 Programmatic Interfaces (APIs).....	78
8.3.6.2 Product Documentation.....	78
8.4 Theory of Operation	79
8.4.1 Socket Driver and Offload Protocol Switch Module Interaction.....	79
8.4.2 Offload Protocol Switch and SDP Module Interaction.....	80
8.4.3 SDP and InfiniBand Transport Interaction	83
9. Architectural Specification – uDAPL	84
9.1 Introduction	84
9.2 Requirement for uDAPL	84
9.3 System Structural Overview.....	85
9.3.1 Component Details.....	88
External interfaces.....	88

Rev. 1.0.1 - 01/Aug/02 01:48 PM

Internal interfaces.....	88
9.3.2 List of External Interfaces.....	88
9.3.2.1 OS Interfaces.....	88
9.3.2.2 Access Layer.....	89
9.3.3 List of Product Deliverables.....	89
9.3.3.1 Consumer Interface.....	89
9.3.3.2 Debug Interface.....	89
9.3.3.3 Dynamo with uDAPL Support.....	89
9.3.3.4 Product Documentation.....	89
9.4 Theory of Operation.....	89
9.4.1 Interface Adapter Creation and Management.....	90
9.4.2 Endpoint.....	90
9.4.3 DAT Connection and Handoff.....	90
9.4.4 Mapping DAPL Connection Qualifier.....	90
9.4.5 Memory Management.....	90
9.4.6 Event Dispatchers.....	91
10. Debugging, Error Logging, and Performance Monitoring.....	92
10.1 Instrumentation.....	92
10.1.1 Performance Counters.....	92
10.1.2 Trace Buffers.....	92
10.2 Debug.....	92
10.2.1 Debug Output.....	92
10.2.1.1 cl_dbg_out and cl_msg_out.....	92
10.2.1.2 Debug Macros.....	93
10.2.2 ASSERT.....	93
10.2.3 cl_panic.....	93
10.2.4 cl_break.....	93
10.3 Error Logging.....	93

Rev. 1.0.1 - 01/Aug/02 01:48 PM

1. Introduction

1.1 Purpose of this Document

This document outlines the structure and major components required to support InfiniBand in the Linux operating system.

The target audience includes software design and sustaining engineers responsible for developing InfiniBand OS and management software. Anyone interested in understanding the high-level software structure of the InfiniBand Linux Software should read this document.

1.2 Document Scope

This SAS presents the structure and behavior of the entire software stack. It only covers the first level of decomposition of the product into its major components. Further decomposition of the major components themselves is included in the High-Level Design, which also includes the APIs.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

1.3 Terminology

1.3.1 Acronyms

Acronym	Description
B_Key	Baseboard Management Key
CI	Channel Interface
CQ	Completion Queue
CQE	Completion Queue Element
CRC	Cyclical Redundancy Check
DGID	Destination GID
EEC	End-to-end Context
EQ	Event Queue
GID	Global Identifier
GMP	General Services Management Packet
GRH	Global Route Header
GSI	General Services Interface
DMA	Direct Memory Access
GUID	Globally Unique Identifier
HCA	Host Channel Adapter
IBA	InfiniBand Architecture
IHV	Independent Hardware Vendor
IPv6	Internet Protocol, version 6
IPoIB	IP over InfiniBand
IOC	I/O Controller
L_Key	Local Key
LID	Local Identifier
LMC	Lid Mask Control
LRH	Local Route Header
M_Key	Management Key
MAD	Management Datagram
MSN	Message Sequence Number
MTU	Maximum Transfer Unit
P_Key	Partition Key
PD	Protection Domain

Rev. 1.0.1 - 01/Aug/02 01:48 PM

PSN	Packet Sequence Number
Q_Key	Queue Key
QoS	Quality of Service
QP	Queue Pair
R_Key	Remote Key
RC	Reliable Connection
RDMA	Remote Direct Memory Access
SD	Sockets Direct
SGID	Source GID
SLID	Source LID
SL	Service Level
SM	Subnet Manager
SMI	Subnet Management Interface
SRP	SCSI RDMA Protocol
SMP	Subnet Management Packet
TCA	Target Channel Adapter
UC	Unreliable Connection
UD	Unreliable Datagram
VL	Virtual Lane
WC	Work Completion
WQ	Work Queue
WQE	Work Queue Element
WQP	Work Queue Pair
WR	Work Request

Table 1-1: Definition of Acronyms used

For additional terminology, see “Appendix A: Definitions”.

1.4 Related Documents

The following documents are referenced by the Software Architecture Specification:

Rev. 1.0.1 - 01/Aug/02 01:48 PM

Document Name	Revision
InfiniBand Architecture Specification	1.0.a
Annex B Sockets Direct Protocol (SDP), Release	1.0.a
IP over IB IETF draft: http://www.ietf.org/ids.by.wg/ipoib.html	
Fast Sockets reference: http://www.cs.purdue.edu/homes/yau/cs690y/fastsocket.ps	
Stream Socket on Shrimp reference: http://www.cs.princeton.edu/shrimp/Papers/canpc97SS.ps	
Memory Mgmt. in User Level Network Interfaces reference: http://www.cs.berkeley.edu/~mdw/papers	
InfiniBand Configuration Management (CFM) Annex E	
SCSI RDMA Protocol – Working Draft, http://t10.org/drafts.htm	

Table 1-2: Related Documentation

Rev. 1.0.1 - 01/Aug/02 01:48 PM

2. Architecture Overview

2.1 Identification

This document describes a software architecture for the Linux operating system and management software components needed to support the InfiniBand Architecture.

It is envisioned that the InfiniBand Linux Software will be developed by the Linux community within the context of a SourceForge project and eventually become part of a standard Linux distribution.

2.2 Goals and Objectives

The goal of the architecture is to describe, in sufficient detail, all of the software components, their responsibilities and behavior, and their interfaces to allow a more detailed design to proceed forward. The architecture will not define the exact APIs. The APIs will be documented in the High Level Design specifications for each software component.

A goal of the architecture is to provide support for multiple vendors HCAs for interoperability.

2.2.1 Compatibility and Interoperability

The architecture employs the use of industry standard interfaces, where available, for communication among the software components. This supports the greatest possible interoperability with third party software. For example, communication between the OS software and the subnet manager will use the standard protocols defined in the InfiniBand Architecture specification. Storage will use the emerging standard SRP protocol and networking will use the emerging standard IPoIB protocol.

2.2.2 Extensibility

The architecture places a requirement on each component requiring the component to provide a mechanism for the user of the interface to determine the revision of an interface.

This allows the user of an interface to determine if it is compatible with a particular implementation and allows the interface to a particular component to be extended over time.

2.2.3 Portability and Platform Independence

The architecture will promote portability to other operating systems wherever possible. However, the architecture is optimized for Linux. Attempts will be made to accommodate the requirements of other more restrictive operating systems, such as EFI, to the extent that it does not result in a sub-optimal implementation for Linux.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

The architecture requires the components support multiple vendor's HCAs and requires processor and platform independence.

2.3 The Architectural Model

The architectural model used to describe the architecture utilizes functional block diagrams.

2.3.1 Architectural Concepts and Representations

In the functional block diagrams that detail the architecture of the various software components, lower level software components (closer to the hardware) are represented on the bottom of the diagram. The level of software abstraction of the hardware typically increases as one moves from the bottom to the top of the diagram.

Distinct functional software blocks are typically displayed as separate boxes in the diagrams. This does not imply or require the software that is described by the block to be implemented as a single software module, but rather is used to delineate the areas where major interfaces will be defined between layers.

Interfaces between components are represented by lines/arrows between components or dashed lines between components. Functional blocks layered on top of each other also imply an interface between the adjacent blocks.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

2.4 Software Architectural Overview

This section provides a description of the overall software architecture and its major components. The section is intended to introduce the reader to the major components of the architecture and only describes the components at a very high level. The chapters that follow provide a more detailed architecture, interfaces, and relationships of each component to the other components and the Linux operating system.

2.4.1 Architecture Block Diagram

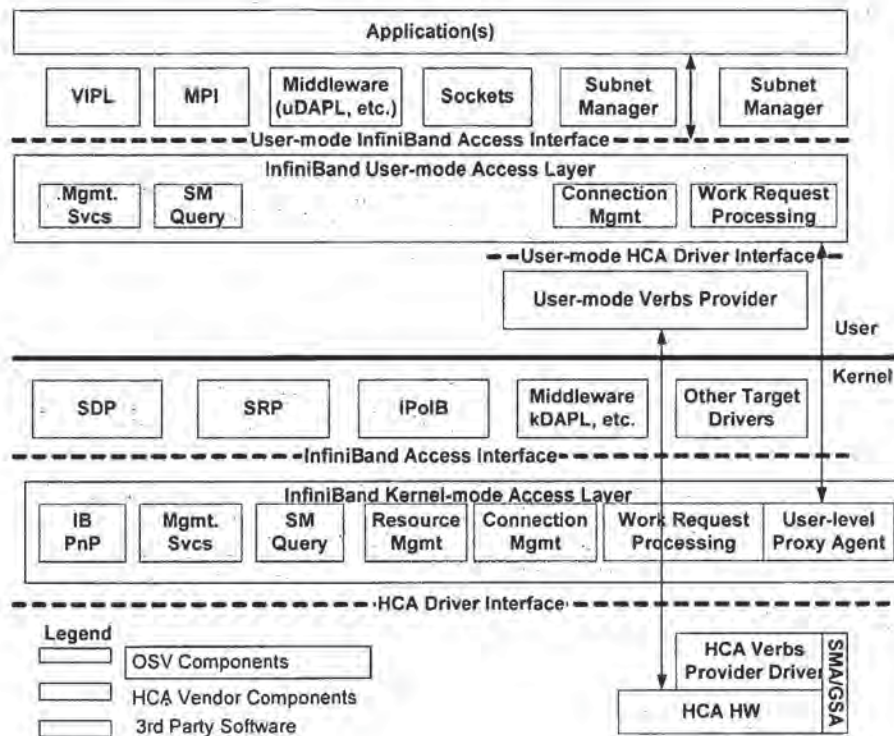


Figure 2-1: Architectural component block diagram

2.4.2 Architecture Block Diagram Description

HCA – Host Channel Adapter provided by a HW vendor.

HW Verbs Provider Driver – This is a vendor-specific piece of software that together with the vendor's HCA forms a unit capable of implementing the verbs as specified in the InfiniBand specification.

HCA Driver Interface – This interface separates vendor-specific HCA code from code that is independent of any particular HCA vendor. The code that sits on top of this interface utilizes the

Rev. 1.0.1 - 01/Aug/02 01:48 PM

functionality of the verbs without being dependent upon any particular vendor's implementation of the verbs.

InfiniBand Kernel-mode Access Layer– This software exports the full capabilities of the underlying HCA implementations to higher-level software. It exists to provide useful services that would otherwise be duplicated in the independent upper-level components, and to coordinate access to shared InfiniBand resources (such as the SMI and GSI) that are needed by multiple upper-level components. The InfiniBand Access component is envisioned to provide several services:

- **Resource Management** – This tracks HCA resource usage by upper level software components. This is useful for cleaning up those resources when/if the upper level components go away, and for having a means of tracking which upper level components should have access to which HCA resources.
- **Work Request Processing** – Process incoming work requests and dispatch the work request to the appropriate HCA drivers and handles the dispatching of work request completion notification to upper-level components. It also notifies interested upper level components of asynchronous events and errors that are detected by the vendor's HCA during its processing.
- **Connection Management** – Encapsulates the connection management protocol as defined in Chapter 12 of Volume 1 of the InfiniBand specification. This significantly simplifies the process of forming connections for those upper-level components that require them, and coordinates access to the GSI amongst those components.
- **SM Query** – Provides an interface to a standard set of subnet administration queries regarding the subnet configuration.
- **Management Services (SMI/GSI QP Access)**– Coordinates access amongst multiple upper-level components to the management queue pairs provided on each port of the vendor's HCA. This also routes incoming SMPs and GMPs to the appropriate upper-level components.
- **User-level Support Services (Proxy Agent)** – Interfaces to the user-mode InfiniBand Access Layer component to support all the services appropriate for user mode that need a kernel transition. Utilizes resources provided by both the host operating system and by the HW specific HCA verbs provider driver to accomplish this. The user-mode InfiniBand Access Layer communicates with the proxy agent through a set of IO control calls.
- **IB PnP** – The plug-and-play implementation for InfiniBand provided by the host operating system.

InfiniBand Access Interface – This exports the interface that all upper-level components within the kernel use to access the functionality provided by an HCA. The Proxy Agent also uses this interface to support some of the services provided to the user mode access layer.

Subnet Manager – Provides the basic Subnet Manager functionality as defined in Volume 1 of the InfiniBand specification.

SDP – Sockets Direct Protocol driver.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

Middleware – Transport and OS independent software layers that support InfiniBand and other RDMA capable transports like iWarp. Also provide an API that is portable across operating systems.

IB Target Drivers – Provides access to TCAs of various types. One example would be a driver provided by the host operating system that utilizes the SCSI RDMA Protocol (SRP) running on top of InfiniBand hardware to access InfiniBand-attached storage. Another example would be a network driver that implements Internet Protocol (IP) over InfiniBand (IPoIB).

User-mode HCA Verbs Provider - Vendor-specific software in user-mode to assist with direct user mode IO (OS bypass) for data transfer operations (DTO).

User-mode HCA Driver Interface - This interface separates user-mode vendor-specific HCA code from code that is independent of any particular HCA vendor.

User-mode Access Layer – This module exports capabilities of all the underlying HCAs that assist in developing upper layer protocols, like VIPL, uDAPL, SM, HCA Diagnostics etc., providing access to InfiniBand primitives in user-mode. The user-mode InfiniBand Access Layer is a vendor independent shared library that can be dynamically loaded by the higher-level software components.

User-mode InfiniBand Access Interface – This is the interface that all applications running in user-mode use to access the underlying HCA. It attempts to minimize the level of abstraction of the underlying hardware while simultaneously being independent of the implementation of any particular vendor's HCA. It also provides the facilities to allow multiple applications to share access the HCAs.

VIPL, MPI, and other messaging interfaces – user-mode implementations of all of these messaging interfaces can be built using the User-mode InfiniBand Access Interface.

Applications – Applications that wish to live “close to the metal” in order to fully exploit the capabilities of the underlying InfiniBand hardware and are willing to use a lower-level interface for optimum performance can access the User-mode InfiniBand Access Interface directly.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

3. Architectural Specification – HCA Verbs Provider Driver

3.1 Introduction

This chapter defines the architecture for the HCA specific, or “verbs provider”, driver software, circled in the figure below. This chapter also contains detailed pictorial representations of the driver software and how it interfaces with other components. The chapter also describes how the driver software interfaces with the hardware, the operating system, and other components of the InfiniBand Linux software stack.

3.2 System Structural Overview

3.2.1 Software Component View

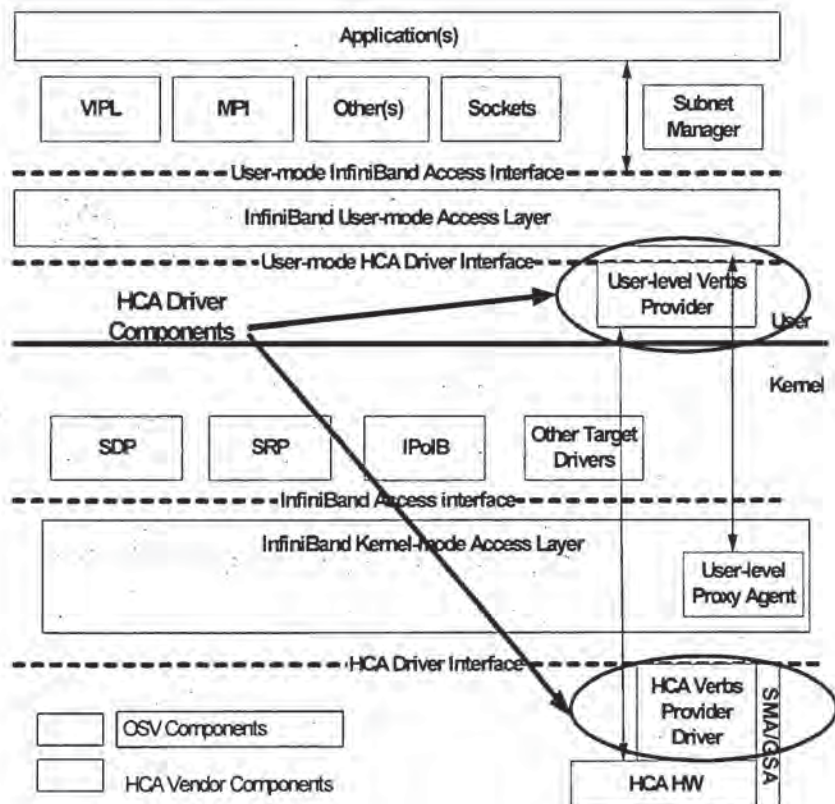


Figure 3-1: Architectural component block diagram

Rev. 1.0.1 - 01/Aug/02 01:48 PM

The HCA Verbs Provider Driver is the lowest level of software in the operating system and interfaces directly with the HCA hardware. The driver software is usually developed and supplied by an HCA hardware vendor. There is typically a separate driver for each different HCA type, although a single driver manages multiple instances of the same HCA type.

The HCA driver has two pieces, a loadable kernel mode device driver (HCA Verbs Provider Driver) and a dynamically loadable user mode library (User-mode Verbs Provider), as shown in the figure above. The user mode library is optional

3.2.2 Relationship between InfiniBand Access and HCA Components

The figures below provide more detail about the interfaces between the HCA driver and the other components in the InfiniBand software stack.

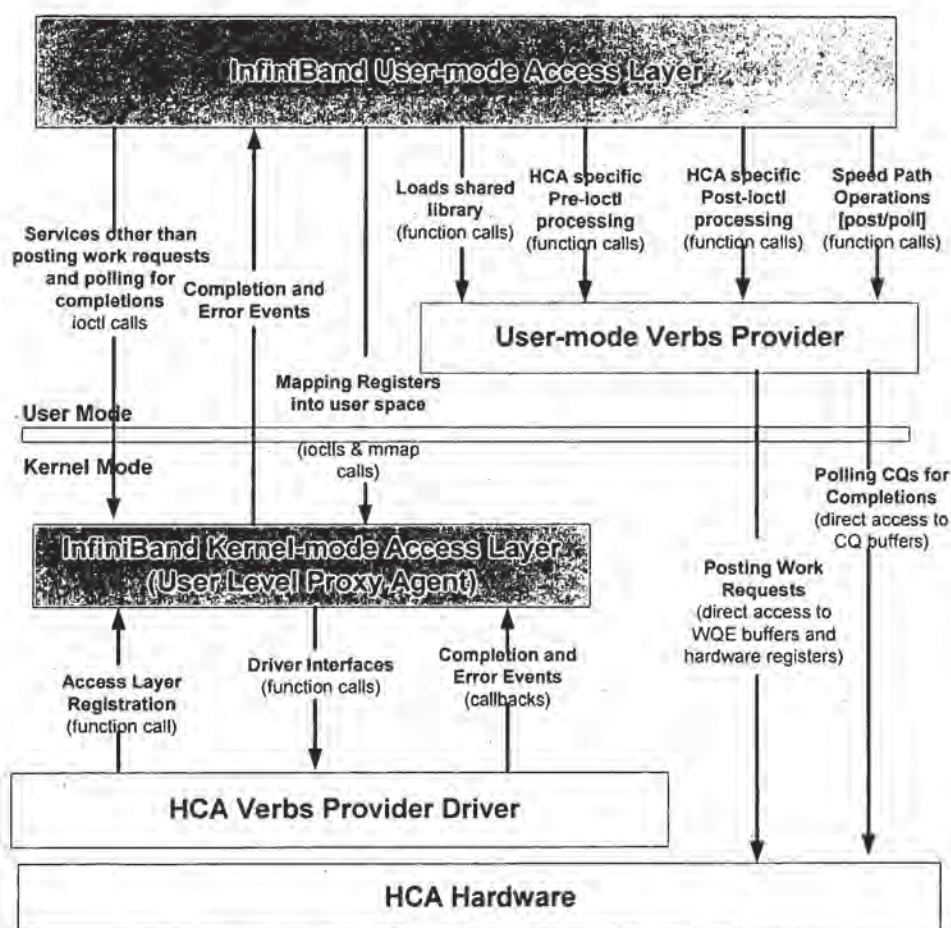


Figure 3-2: Interfaces between InfiniBand Access and HCA Software

The user mode library plugs into the InfiniBand User-mode Access Layer. The User-mode Access Layer discovers it by an entry in a system configuration file that associates a library with

Rev. 1.0.1 - 01/Aug/02 01:48 PM

the HCA vendor. If a User-mode Verbs Provider is not provided for a HCA type, the User-mode Access Layer will use ioctl calls to the InfiniBand Kernel-mode Access Layer for all functions. However, if a user mode component is provided, it provides access to certain APIs for the major speed path operations. The architecture allows a vendor to provide side band access to aid direct communication with the hardware for speed path. An example would be posting work requests and polling completions. In this case, the InfiniBand User-mode Access Layer calls the HCA User-mode Verbs Provider directly rather than making an ioctl call to the proxy agent in the kernel for those speed path operations.

If a User-mode Verbs Provider is available for the HCA type, the InfiniBand User-mode Access Layer will call the HCA User-mode Verbs Provider twice for each verbs related function that needs a kernel transition. The first call will be made before the ioctl to the kernel and the second one after the ioctl returns from the kernel. The HCA User-mode Verbs Provider can do any resource initialization and clean up during these calls. For e.g. it can map the H/W doorbell registers into the user process's virtual address space and allocate buffers for WQE construction and completion queues. The InfiniBand User-mode Access Layer will also provide a mechanism during these calls to pass private data between the HCA User-mode Verbs Provider and kernel mode HCA Verbs Provider. Once the necessary resources are set up and accessible from the user space, the User-mode Verbs Provider then communicates directly with the hardware for posting work requests. HCA hardware then fetches and processes the work request. After the HCA processes the work request it posts the completion in the CQ buffer in the process address space. The notification is then generated by the HCA if the CQ was enabled for notifications by writing an entry to the Event Queue. The InfiniBand User-mode Access Layer then notifies the user process about the completion. The user process can then poll the completion queue to obtain status of the work request submitted.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

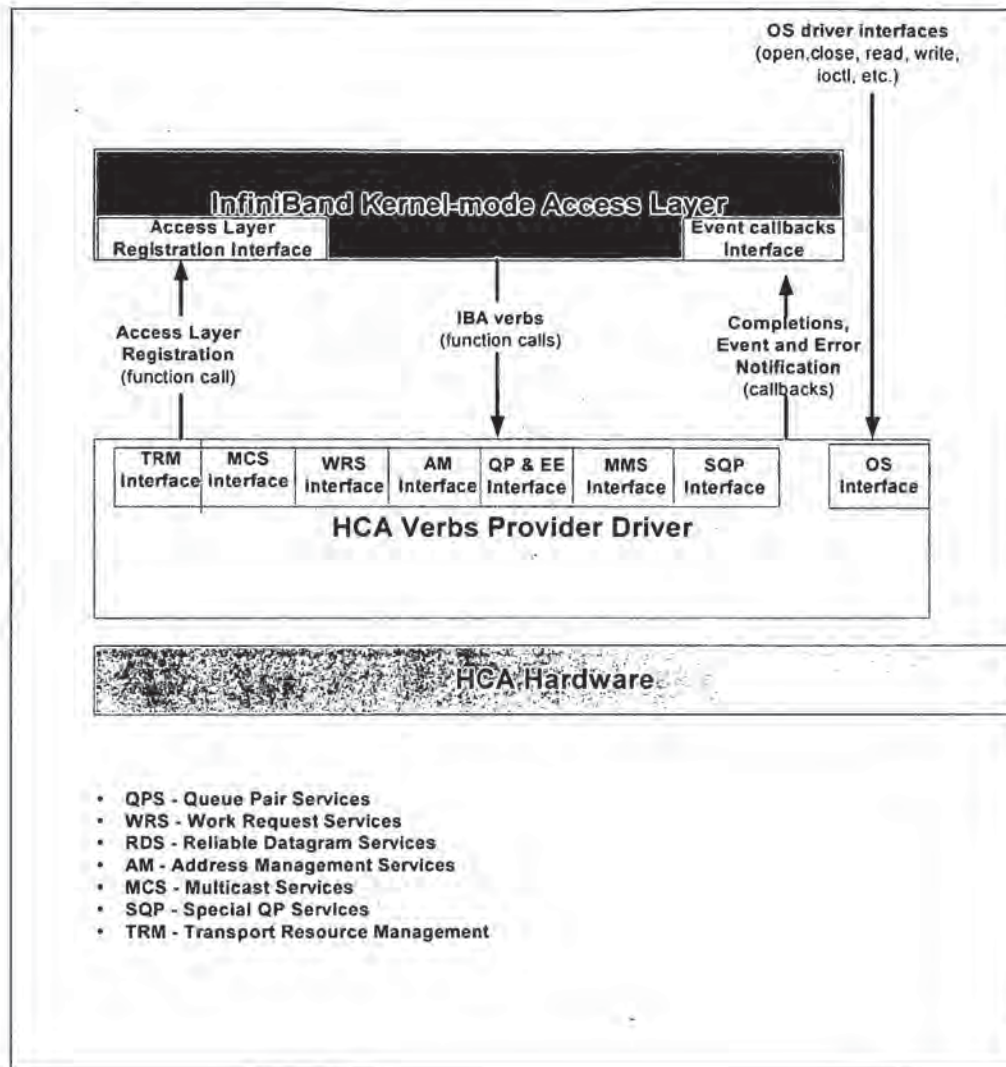


Figure 3-3: HCA driver software components

The figure above provides more detail on the interfaces provided by the kernel mode HCA Verbs Provider driver. The HCA driver interfaces with the Linux kernel as a regular Linux PCI device driver.

The kernel mode driver interfaces directly with the HCA hardware on its bottom end. It discovers the presence of the HCA hardware as a regular Linux PCI driver. The PCI kernel mode component provides the HCA driver with necessary information about location and size of registers, interrupt vector etc, using the existing mechanisms provided by the Linux OS for any device driver.

The InfiniBand Kernel-mode Access Layer provides an API to allow HCA drivers to register with the Access Layer during HCA driver initialization. The Kernel-mode Access Layer then uses the direct function calls exported by the HCA driver to get access to the Verbs functionality.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

Chapters 10 and 11 of the InfiniBand Specifications Volume 1.0a provide a list of the functionality that the driver is supposed to provide. The major functional groups providing access to the hardware are shown below

API Group	Functionality
Transport Resource Management	HCA access, Obtain HCA resources such as protection domains, reliable datagram domains
Address Management	Create, modify, query and destroy address handles
Queue Pair and EE Context Management	Create, modify, query and destroy queue pairs and EE contexts.
Special Queue Pair Operations	Allocate Special Queue Pair for QP0, QP1, Raw, Raw Ether queue pair types.
Memory Management	Register virtual, physical address, managing memory windows.
Multicast services	Attaching and detaching a QP from a multicast group
Work Request Processing	Posting work requests to send and receive queues. Polling for completion queue entries, requesting for completion notification.
Event Handling	Ability to notify consumers for completion and asynchronous events.

Table 3-1: Collection of Verbs Groups

Rev. 1.0.1 - 01/Aug/02 01:48 PM

3.3 Theory of Operation

3.3.1 Driver Interfaces

This section provides details on the interfaces provided by the and their capabilities. A later section presents a typical usage model of how the Access Layer uses the interfaces. The majority of the interfaces to the driver are derived from the Transport Verbs chapter of the InfiniBand architecture specification V011a. It is assumed that the driver will have nearly a 1 to 1 mapping of API calls to the verbs defined by InfiniBand.

3.3.1.1 Driver initialization Interfaces

The HCA Verbs Provider driver provides interfaces that allow the InfiniBand Kernel-mode Access Layer to open and initialize the driver, and determine the capabilities of the HCA.

Refer to the InfiniBand specification for details on the specific capabilities provided by these interfaces.

In Linux, the HCA Verbs Provider driver is a loadable kernel module that typically gets loaded during system start-up. The InfiniBand Kernel-mode Access Layer driver is loaded prior to the HCA Verbs Provider drivers. This is traditionally controlled via module dependencies and load order specified in the */etc/modules.conf* file.

If the HCA presents itself as a PCI device to the hardware, the driver registers with the PCI device driver. It provides the PCI device-id/vendor-id to the PCI driver. When the PCI driver finds an HCA that matches the PCI vendor-id/device-id, it calls the driver back providing pointers to the HCA resources. These are the typical interfaces used in Linux for PCI devices.

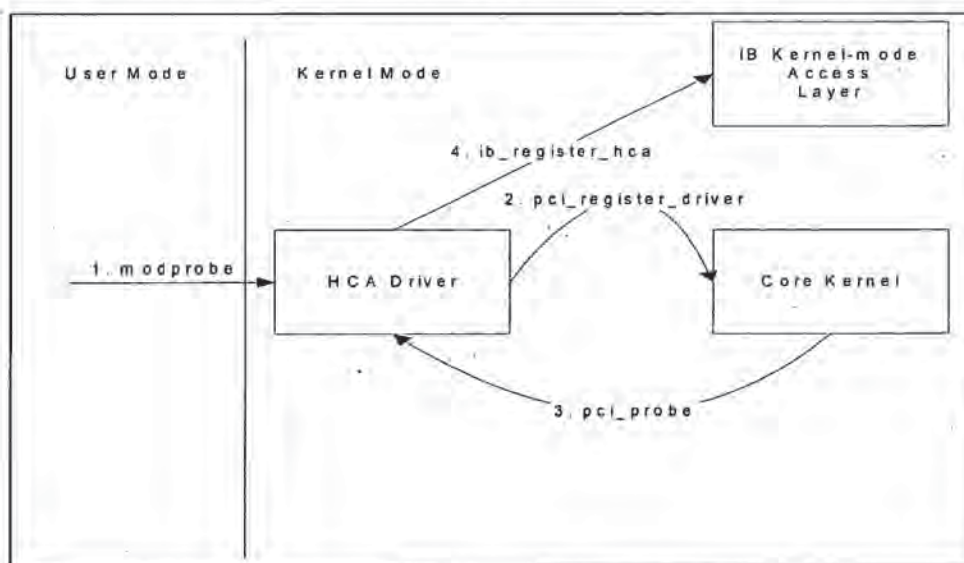


Figure 3-4: Loading the HCA driver

Rev. 1.0.1 - 01/Aug/02 01:48 PM

3.3.1.2 Driver close, unload, and cleanup

The HCA Verbs-provider driver notifies the InfiniBand Kernel-mode Access Layer when the driver is shutting down. This is triggered when a HCA hot-remove happens. The HCA Verbs Provider driver then calls the InfiniBand Kernel-mode Access Layer to de-register the HCA leaving the system. The Kernel-mode Access layer must remove all references to the specified HCA before the de-register API call completes.

Dynamically loaded drivers are protected from unloading by maintaining reference counts. The driver *cleanup_module()* call must never fail. Hence it is important to maintain appropriate reference counts on each *open_hca()* call from the InfiniBand Kernel-mode Access Layer.

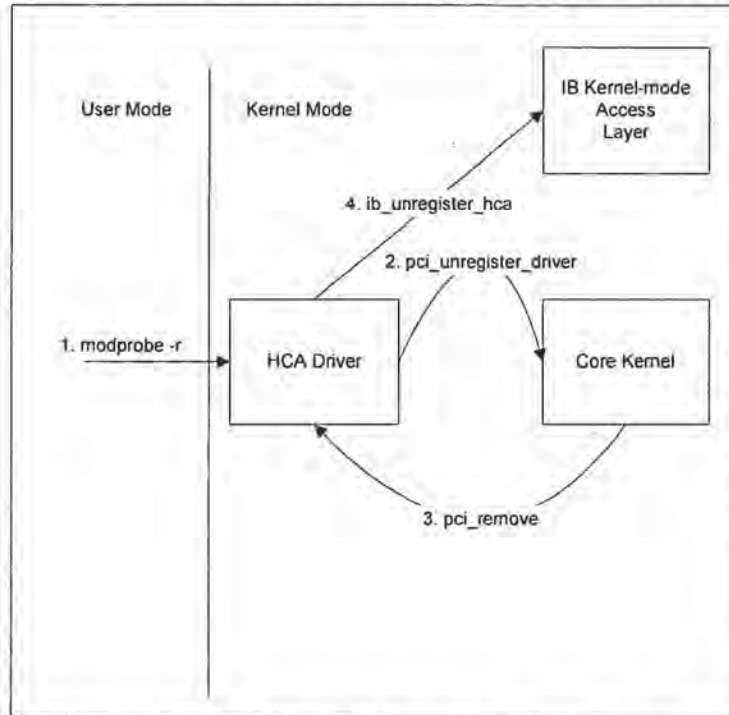


Figure 3-5: HCA Driver Unload Sequence

3.3.2 Typical Driver Usage Model

This section describes a typical usage model of the driver interfaces by the InfiniBand Kernel-mode Access Layer. It does not define every possible usage model, but outlines the way the driver interfaces could be used in a normal operation.

Initialization

The HCA driver is a loadable kernel module that typically gets loaded during system start-up. It is loaded sometime after the InfiniBand Kernel-mode Access Layer.

If the HCA presents itself as a PCI device to the hardware, the driver is instantiated by the Linux PCI driver. Upon start-up, the driver performs any required HCA initialization and sets up and initializes the driver data structures, e.g., things like the event queue, TPT, etc.

Next, the driver registers with the InfiniBand Kernel-mode Access Layer providing the GID of the HCA. The Kernel-mode Access Layer provides an interface to allow the driver to perform this registration.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

The Kernel-mode Access Layer can then open and query the driver using the GID to get the resource attributes of the HCA, such as the number of QPs, CQs, TPT, etc. The Kernel-mode Access Layer can then also register a callback for asynchronous event notifications. This allows the Kernel-mode Access Layer the ability to process events that are not associated with a given work request. E.g., port state events.

Queue Pair Setup in preparation for sending data

Before the InfiniBand Access Layer can send data across the fabric, it must allocate protection domains, set up queue pairs and completion queues, and establish connections with remote nodes.

A protection domain must be allocated using the Get Protection Domain interface so that it can be associated with the queue pair when it is created. The Access Layer or upper layers of software can allocate a separate protection domain for each of its clients or client processes to provide memory protection between clients and/or processes. If the connection that the access layer is intending to create is a reliable datagram, the access layer must allocate a reliable datagram protection domain.

After obtaining a PD, the Access Layer creates a completion queue to be associated with the work queue pair. This is done using the Create Completion Queue Interface. Next the Access Layer allocates the queue pair using the Create Queue Pair interface passing the CQ, PD, and other desired attributes of the QP. The connection manager component of the Access Layer can then use the Modify QP interface to modify the state of the queue pair during connection establishment.

Memory Registration in preparation for sending data

After the queue pairs have been allocated and set up for communication, the Access Layer must register all memory buffers that contain the WQEs or the user data buffers to/from which data will be transferred. To accomplish this, the Access Layer uses the memory registration interfaces of the Verbs Provider driver.

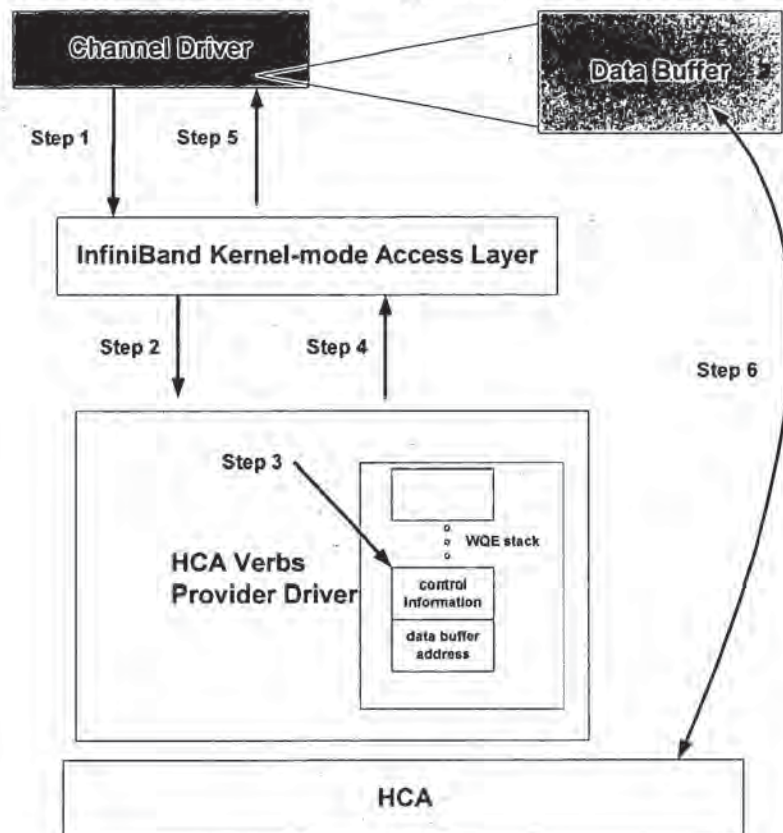
The Register Memory Region interface is used to register virtual addresses and the Register Physical Region interface is used to register physical memory regions. The Access Layer can use the Register Shared Memory Region interface to register memory that is shared between protection domains. It can use the memory window routines to allocate a memory window and then later bind the memory window to a memory region using a work request, but no matter which registration mechanism is used, all memory buffers that are described in work requests must be registered.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

Posting Work Requests

Once the Access Layer has allocated and initialized queue pairs and completion queues and registered the memory buffers associated with a data transfer, the Access Layer uses the Post Send and Post Receive interfaces to post work requests.

The diagram below illustrates the typical flow of posting a send or receive work request.



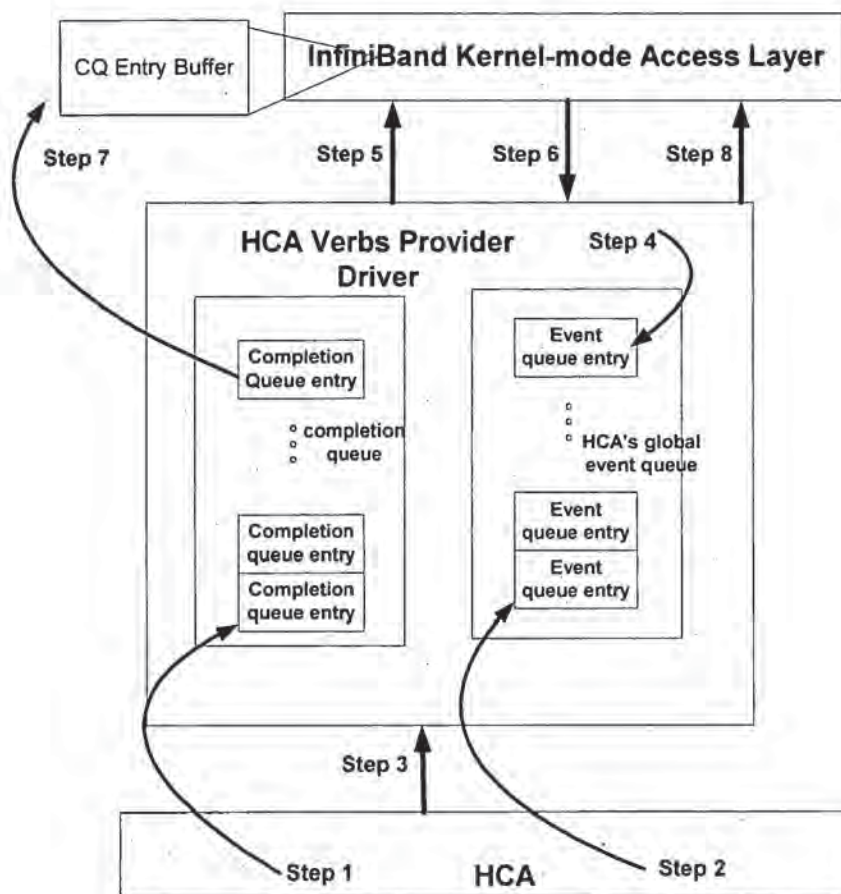
1. Channel Driver posts Work Request to Kernel-mode Access Layer.
2. Kernel-mode Access Layer Posts the Work Request to the appropriate driver.
3. HCA driver builds WQE, posts it to the H/W queue, and rings the doorbell.
4. HCA Verbs-Provider Driver returns to the Kernel-mode Access Layer.
5. Kernel-mode Access Layer returns to the Channel Driver.
6. HCA transfers data to/from the data buffer asynchronously while processing the WQE.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

Processing Completions

Once work requests have been posted to the Send or Receive queues and the H/W has completed the work requests, the Access Layer can process the associated completions.

The diagram below shows the typical flow of the completion processing.



1. HCA fills in completion queue entry upon completion of a work request.
2. HCA fills in event queue entry.
3. HCA sends an interrupt to the HCA Verbs Provider driver.
4. The HCA driver polls the event queue to determine the type of event.
5. The HCA driver calls the Kernel-mode Access Layer's completion callback function
6. The Kernel-mode Access Layer Calls the Poll CQ routine.
7. The driver copies the CQ entry to the Access Layer buffer
8. The HCA Verbs Provider driver returns from the Poll CQ call.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

Memory Deregistration

Once the work requests have completed, the Access Layer can use the Deregister memory interfaces to deregister the memory buffers. This frees up the TPT entries for use in subsequent memory registration operations. It is not uncommon for memory buffers to be registered and deregistered for each I/O operation, but it is possible for the Access Layer or upper layers of software to reuse the same I/O buffers and thus only need to register the memory once. This can lead to increased performance in some applications.

Destroying Queue Pairs and Cleaning up after Disconnections

Typically when a connection with a remote node is terminated, the Kernel-mode Access Layer will release the resources associated with that connection for use in subsequent connections. The Access Layer uses the Destroy Queue Pair interface to free the queue pair for subsequent use. It uses the Destroy Completion Queue interface to release the CQ resources. It uses Deallocate Protection Domain and Deallocate Reliable Datagram Domain to free the PD resources that were associated with the QP.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

4. Architectural Specification – Access Layer

4.1 Introduction

The access layer provides transport level access to an InfiniBand fabric. It supplies a foundation upon which a channel driver may be built. The access layer exposes the capabilities of the InfiniBand architecture, and adds support for higher-level functionality required by most users of an InfiniBand fabric. Users define the protocols and policies used by the access layer, and the access layer implements them under the direction of a user.

4.2 Design Methodology

The access layer architecture is constructed using bottom-up, object-oriented design techniques with top-down functional requirements. Based on user requirements, specific functionality exposed by the access layer is captured. Commonality between the different functional areas is then identified, and generic components are constructed to support the common areas. Generic components are created using standard library modules where possible. To support user-specific API requirements, detailed modules are layered over the generic components. This is shown in the Figure 4-1 below.

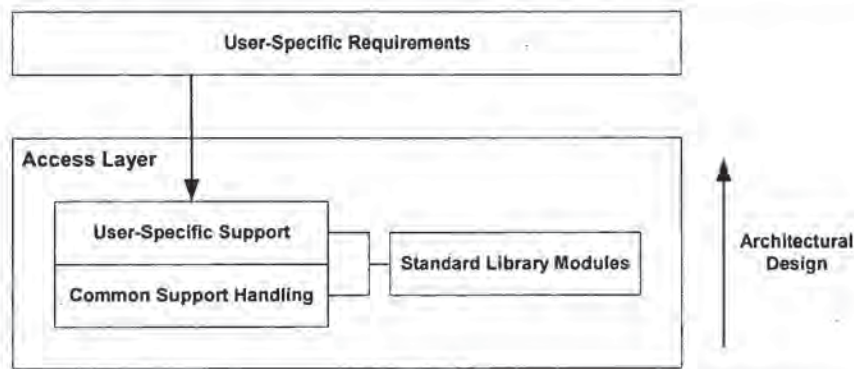


Figure 4-1: Access Layer Architectural Design

4.2.1 Functional Services

The Figure 4-2 highlights the services provided by the access layer.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

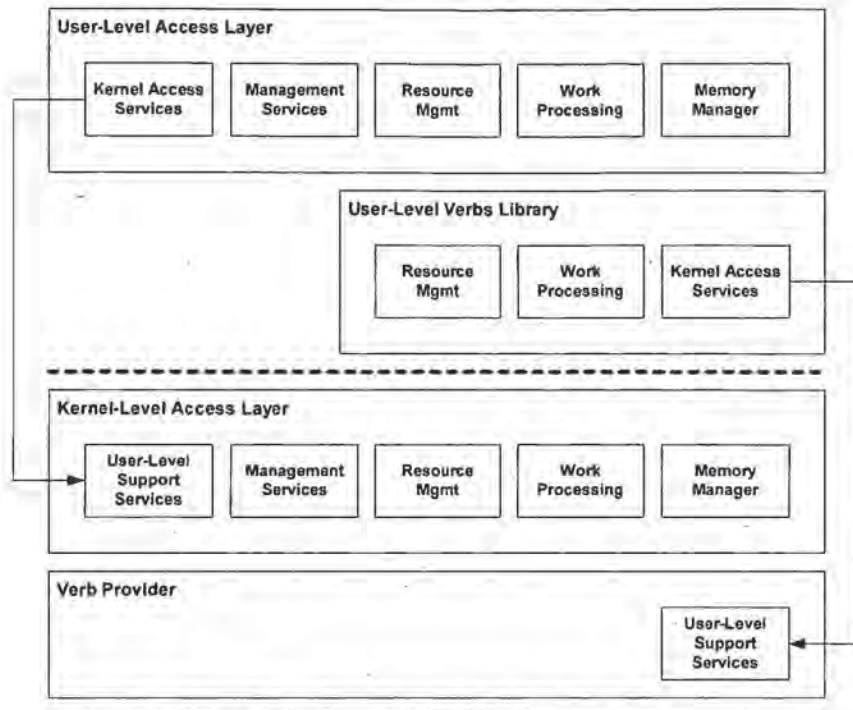


Figure 4-2: Access Layer Functional Services

As seen in the figure, the access layer is located above the verbs provider drivers and operates in both kernel and user-level environments. Kernel access and user-level support services provide proxy agents and implement IO control codes to communicate between the user-level and kernel-level modules. The user-level verbs library is an optional component that provides a speed path for some operations. If a user-level verbs library is not provided, the user-mode access layer will be used for all user-level operations. The main features of the access layer are grouped into the following areas: management services, resource management, work processing, and memory management. These areas are discussed in more detail in the following sections.

4.3 Theory of Operation

4.3.1 Management Services

The access layer provides extensive support for sending, receiving, and processing management datagrams (MADs). All management classes are supported. Figure 4-3 illustrates the MAD architecture exposed by the access layer.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

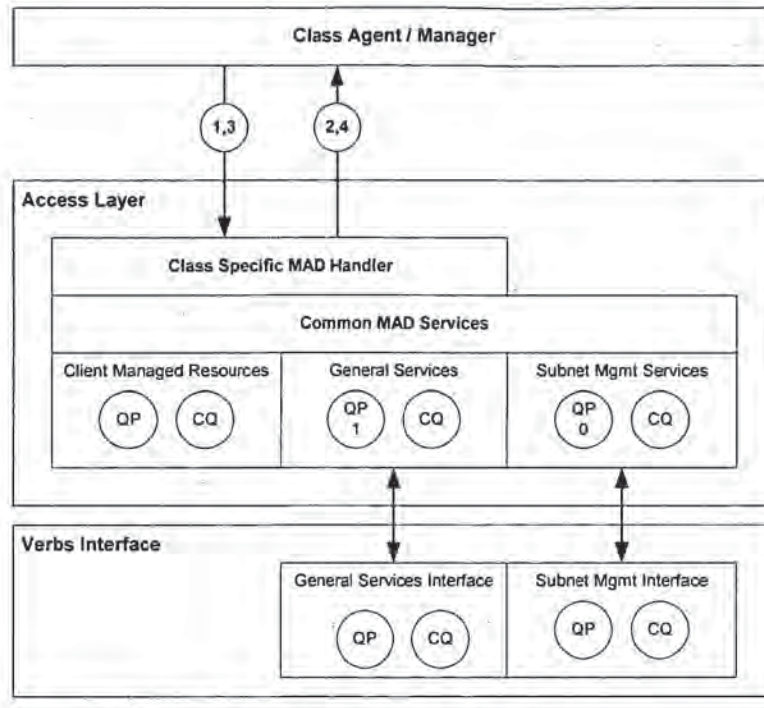


Figure 4-3: Access Layer MAD Architecture

Operation:

1. Clients register as an agent or manager of a specific class. All management classes are supported, but only one class manager may be registered for a specific class.
2. The access layer returns an alias to a QP that may be used to send management datagrams.
3. The class agent or manager posts MADs to the QP alias or, in the case of QP redirection, their own QP.
4. Clients receive incoming MADs through a callback. Unsolicited MADs are routed to the registered class manager. Responses are directed to the agent who initiated the request.

As shown by Figure 4-3, the access layer supports both subnet management (QP0) and general service (QP1) datagrams. Work requests posted to QP0 or QP1 are always processed by the access layer; direct access to these QPs is not allowed. MAD services are divided between common and class specific services. Class specific services deal directly with MADs for a given management class, such as communication management. For convenience, class specific handlers support high-level helper functions that allow users to avoid dealing directly with MADs as shown in Figure 4-4. Class specific helper functions are described in section 4.3.1.1.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

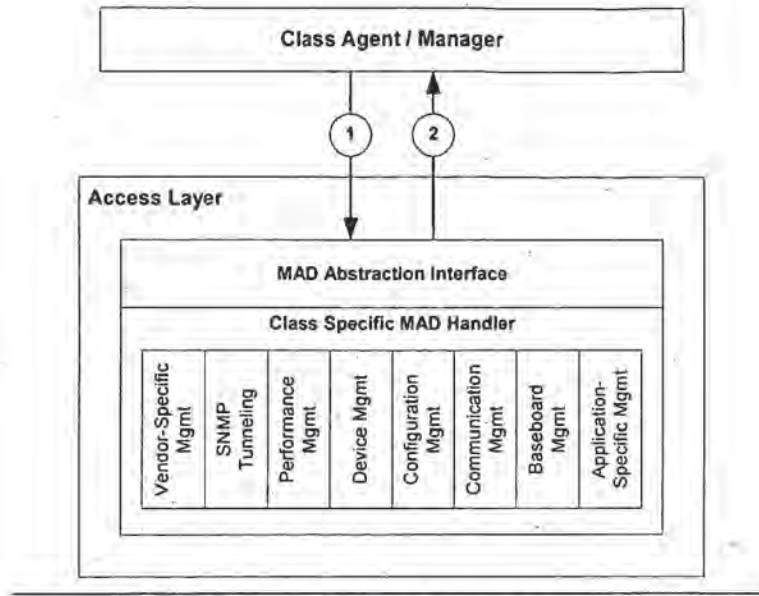


Figure 4-4: MAD Class Specific Handlers

Operation:

1. A user invokes a helper function supported by one of the class specific handlers. MAD helper functions format and send MADs on behalf of the user.
2. The response to a user's request is returned asynchronously to the user.

Regardless of a MAD's class, the access layer provides the following features where appropriate.

- Transaction ID management
- Segmentation and reassembly (SAR) for MAD classes that require SAR
- Support for queue pair redirection
- Queuing to ensure that multiple clients do not overrun an underlying queue pair
- Retransmission of MADs for which a response is expected
- Disabling of MAD support on a per-port basis

4.3.1.1 Management Helper Functions

The access layer provides a set of helper functions that allow users to perform MAD transactions without dealing directly with MADs themselves. These functions include Service Record Registration, Service ID Registration and Resolution, Information Queries, Registration For Local Events, and Subscription For Reports.

4.3.1.1.1 Service Record Registration

Service records allow clients to advertise the availability of basic services to the subnet. The access layer provides support for clients to register and de-register service records with subnet administration. The client provides the service record

Rev. 1.0.1 - 01/Aug/02 01:48 PM

with the registration request. The access layer returns a handle to the client and performs the MAD transactions with subnet administration. Using the returned handle, the client can unregister the service. The result of the service registration or de-registration is returned to the client through a callback function.

4.3.1.1.2 Service ID Registration and Resolution

Registration of a service ID provides a method for remote users of unreliable datagram services to determine the queue pair number and queue key of a port that supports a given service ID. The access layer provides an interface to allow clients to register and deregister a service ID with the communication manager and to resolve service IDs using the service ID resolution protocol.

Registration:

A client registers a service ID by providing the service ID, the channel adapter and port GUIDs, a callback function, and an optional buffer and length for comparing the private data exchanged during the service ID resolution process. The access layer returns a handle that the client may use to deregister the service.

Resolution:

A client requesting service ID resolution provides the desired service ID, a path record to the remote port, a callback function, and an optional buffer containing private data. A high-level view of the service ID resolution process is shown in Figure 4-5.

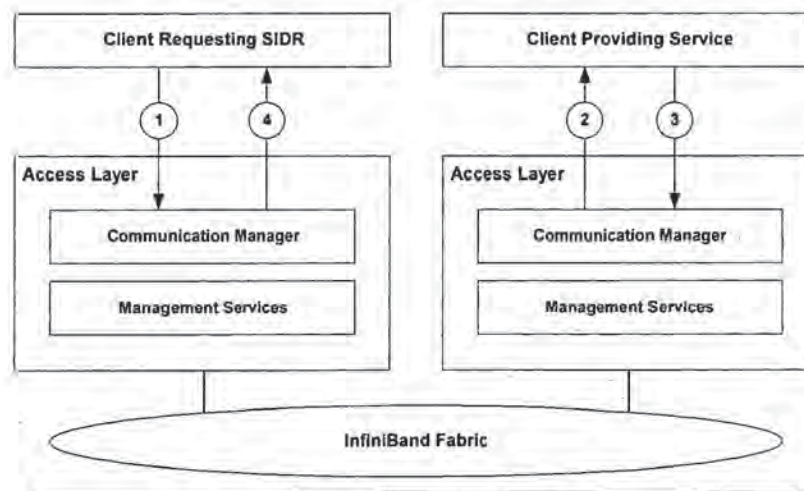


Figure 4-5: Service ID Resolution

Operation:

1. A client makes a service ID resolution request. The communication manager sends a service ID resolution request MAD to a remote node.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

2. The communication manager invokes a callback after receiving a service ID resolution request. The client is given information about the request including the private data.
3. The client accepts or rejects the service ID resolution request. A client accepts by providing the QP and QKey. A rejection may optionally indicate redirection to another port.
4. The communication manager returns the result of the service ID resolution process.

4.3.1.1.3 Information Queries

The access layer provides support for client to query for commonly used data. Based on the query type and input data, the access layer performs required the MAD transactions with the appropriate class management entities to obtain the requested data. The result of the query operation is provided to the client through a callback function.

The access layer provides the following queries:

- Query for service records by service name
- Query for service records by well-known service ID
- Query for node record by node GUID
- Query for port record by port GUID
- Query for path records by a port GUID pair
- Query for path records by a GUID pair

The access layer supports other queries by sending and receiving MADs through the standard management services.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

4.3.1.1.4 Registration For Local Events

Clients of the access layer can register for notification of local events. Local events include the insertion and removal of channel adapters, ports becoming active and inactive, LID changes, partition changes, and IO controllers being assigned or unassigned to a port. Figure 4-6 illustrates the operation of local events.

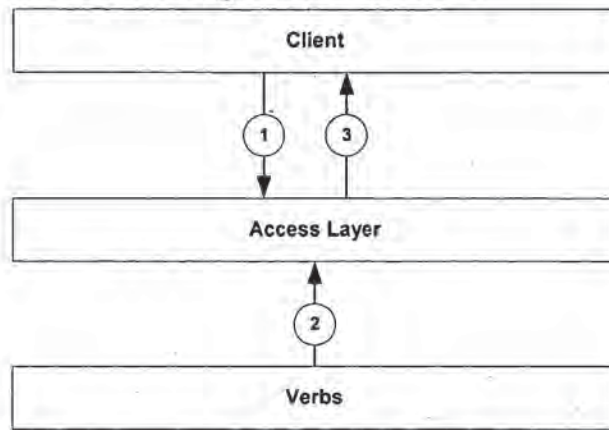


Figure 4-6: Local Event Operation

Operation:

1. The client registers for notification of a local event.
2. The access layer receives notification of a locally occurring event.
3. The access layer notifies clients registered for the local event via a callback.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

4.3.1.1.5 Subscription For Reports

Clients of the access layer can register for reports delivered through the event forwarding mechanism.

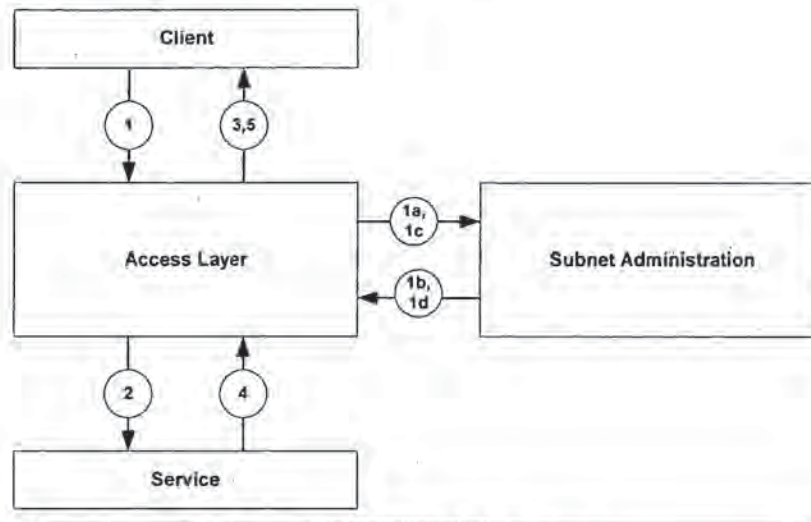


Figure 4-7: Report Subscription

Operation:

1. The client subscribes for a report providing the InformInfo record and the name of the service to subscribe with. The access layer performs the following tasks:
 - a. The access layer issues a query to the subnet administration for the named service record. Refer to section 4.3.1.1.3
 - b. The access layer receives the requested service record. The service record contains the GID where the specified service may be found.
 - c. The access layer issues a query to the subnet administration to obtain path records associated with the specified service using the source and destination GIDs. Refer to section 4.3.1.1.3.
 - d. The access layer receives the requested path records.
2. The access layer initiates a MAD transaction to subscribe for the report.
3. The access layer returns the result of the subscription process to the client through a callback.
4. At some point in the future, the access layer receives a report from the service for the subscribed event.
5. The access layer notifies the client of the subscribed event through a report callback function.

4.3.1.1.6 Rejecting An IO Controller Assignment

The InfiniBand Annex E: Configuration Management specification allows a host to reject the assignment of an IO controller. The access layer can perform this operation on behalf of a client. The client requests the IO controller rejection

Rev. 1.0.1 - 01/Aug/02 01:48 PM

providing the reported IOC information. The access layer performs the MAD transactions with the configuration manager.

4.3.1.2 Communication Manager

In addition to providing support for class managers and agents to send and receive MADs, the access layer provides a class manager for communication. The communication manager abstracts connecting a queue pair with a remote queue pair and transitions the queue pair through the proper states. A high-level view of the communication manager is shown in Figure 4-8.

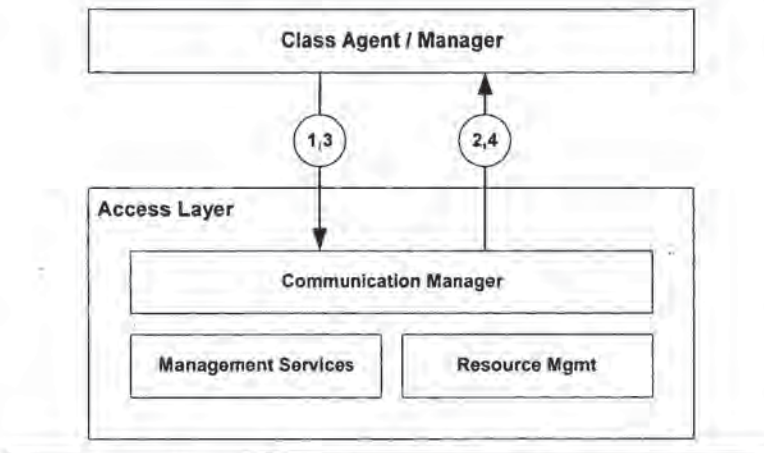


Figure 4-8: Communication Manager

Operation:

1. Clients make a request to connect a QP. For active connection requests, this results in the communication manager sending a connection request MAD to a remote node.
2. The communication manager invokes a callback after receiving a reply to the connection request. The user is given information about the connection attempt, along with the queue pair that will be used.
3. The client accepts or rejects the connection.
4. The communication manager returns the result of the connection process. If the connection was successful, the client may perform data transfers on the given queue pair.

The communication manager performs the following functions.

- Transitions the queue pair state throughout the connection process
- Manages the state of the connection throughout its lifetime
- Supports automatic path migration on the connection

Rev. 1.0.1 - 01/Aug/02 01:48 PM

4.3.1.3 Device Management Agent

Some clients of the access layer may require the ability to export an IO controller to the InfiniBand fabric as an IO unit. To support this requirement, the access layer provides a device management agent. The device management agent provides a central point of operation that allows multiple clients to register IO controllers. The device management agent maintains the data necessary to respond to device management MADs related to IO unit operations. A high-level view of the device management agent is shown in Figure 4-9.

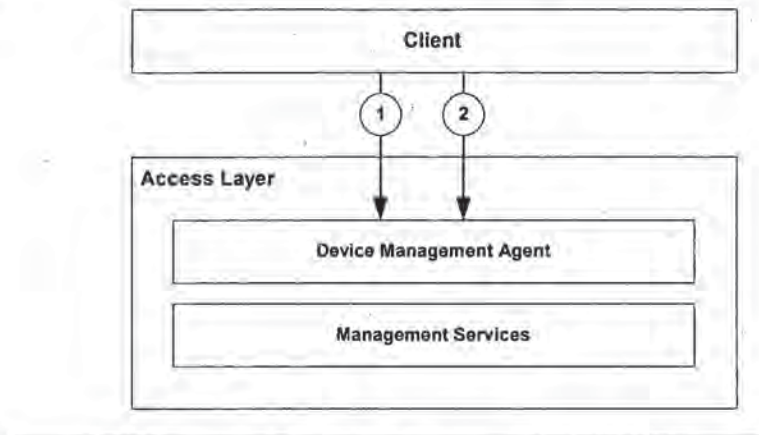


Figure 4-9: Device Management Agent

Operation:

1. The client registers an IO controller profile with the device management agent for a channel adapter. The first IO controller profile registered causes the access layer to set the PortInfo CapabilityMask IsDeviceManagementSupported flag for all ports of the given channel adapter. The access layer returns a handle to the registered IO controller.
1. The client adds service entries to the registered IO controller. The access layer creates a new service entry for the IO controller and returns a service entry handle.

The device management agent performs the following functions.

- Maintains the IO unit information, IO controller profiles, and service entries on behalf of all registered clients
- Automatically assigns IO controllers to slots.
- Indicates the presence of device management support in the PortInfo CapabilityMask flags
- Responds to the device management class MADs related to IO unit operations

4.3.1.4 Plug and Play Manager

The purpose of the plug and play manager is to notify kernel-mode clients of changes in device status. For example, a channel driver may request notification when an IO

Rev. 1.0.1 - 01/Aug/02 01:48 PM

controller becomes available or unavailable, or perhaps when a channel adapter port becomes active or goes offline.

The plug and play manager reads a configuration file that contains a mapping of notification events to channel drivers. If a channel driver is not present at the time the plug and play manager needs to notify it, the plug and play manager will load the driver. The plug and play manager delivers notifications to channel drivers asynchronously through a callback registered. Figure 4-10 illustrates the channel driver load and notification process.

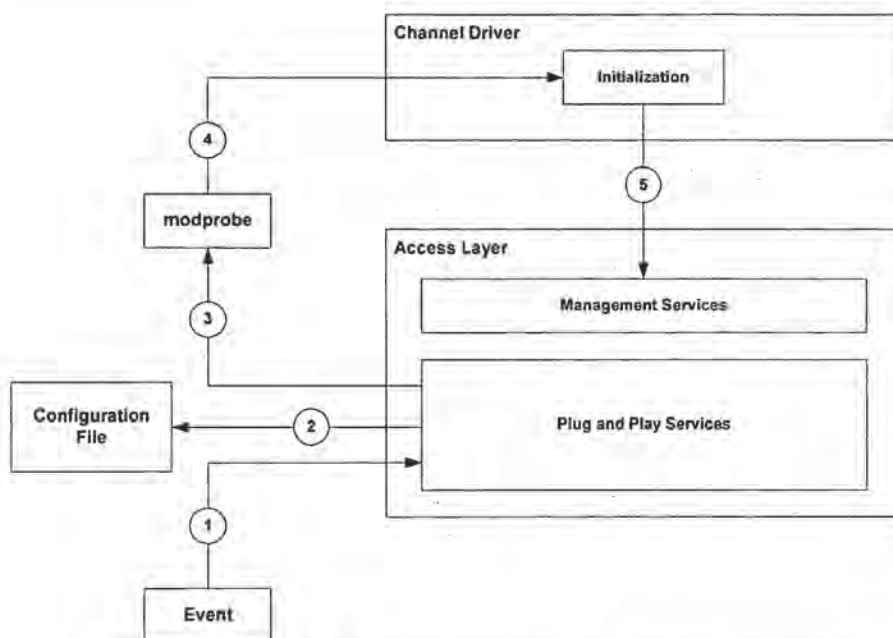


Figure 4-10: Channel Driver Load Sequence

1. The plug and play manager receives notification of an event that has occurred, such as the addition of a new IO controller.
2. The plug and play manager reads a configuration file to determine the correct channel driver to load.
3. The plug and play manager invokes a user-level application to load the appropriate channel driver.
4. The channel driver module is loaded and the driver initialization function is called.
5. The channel driver opens the access layer.

The plug and play manager is built upon the local event and report subscription mechanisms described in 4.3.1.1.4 and 4.3.1.1.5. The notification callback uses the same

Rev. 1.0.1 - 01/Aug/02 01:48 PM

parameter format as the local event and report subscription callbacks. Preserving this interface allows the configuration file to be thought of as a “manual” method for a channel driver to subscribe for notifications before the channel driver is actually loaded. The configuration file format supports a subset of the notification capabilities provided by the programmatic interface.

For IO controller channel drivers, the plug and play manager uses the compatibility string matching method described in InfiniBand Annex A: IO Infrastructure specification. This technique allows the configuration file to list a set of compatibility strings for each driver. The matching algorithm uses a search order that selects more specific channel drivers over more generic channel drivers.

4.3.2 Resource Management

The resource management support provided by the access layer is responsible for the allocation and management of exposed channel adapter resources, such as queue pairs, completion queues, address vectors, and so forth. The resource manager is responsible for managing resource domains for resources that are allocated within a hierarchy. For example a CA resource contains CQ and PD resources, and PD's contain QPs. Destruction of a resource domain will automatically result in the release of all associated resources. Figure 4-11 shows the general operation of the resource manager.

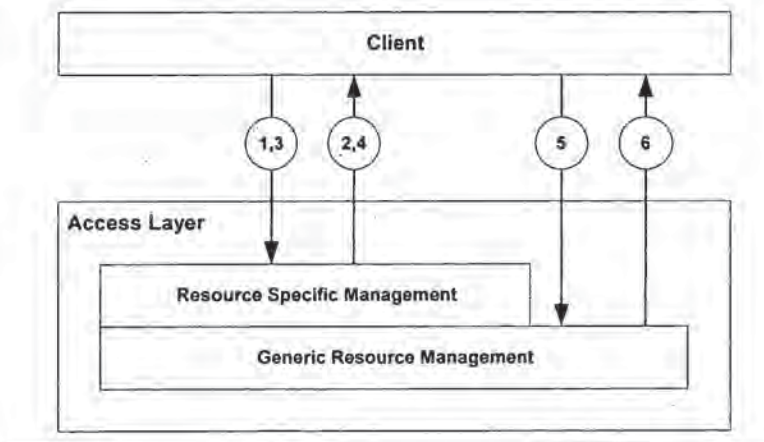


Figure 4-11: Resource Manager

Operation:

1. Clients allocate a new resource by invoking a resource specific function.
2. The access layer synchronously returns a handle to the allocated resource.
3. Clients can query the access layer for additional properties of the resource. The type of information returned is dependent on the resource being accessed.
4. The access layer synchronously returns the properties of the resource.
5. Once a resource is no longer needed, it may be destroyed through a destruction call.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

6. Once the resource has been destroyed, the user is notified. To avoid race conditions, the destruction process operates asynchronously.

Resource management performs the following functions.

- Expose all resources accessible through verbs.
- Manage resource domains to permit destruction of an entire domain. For example, destroying an instance of a CA will automatically release all related resources, such as CQs, QPs, etc.
- Resource management is not considered a critical speed-path operation.

4.3.3 Work Processing

The work processing module handles issuing work requests and completion processing. In general, work processing deals with operations that occur on queue pairs and completion queues.

4.3.3.1 General Work Processing

A general overview of work processing operation is shown in Figure 4-12.

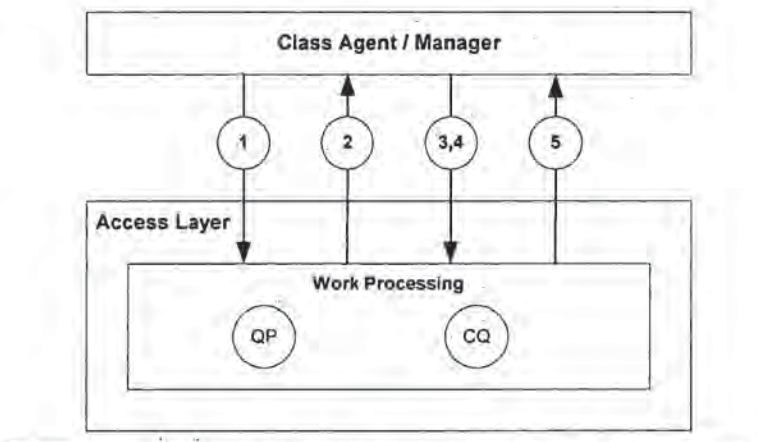


Figure 4-12: Work Processing Overview

Operation:

1. Clients post a work request to a specified queue pair.
2. Upon completion of a work request, the completion queue signals an event to notify the user of the completion. Clients may either wait on a completion event or may be notified through a callback mechanism.
3. Clients re-arm the completion queue to indicate that the completion event should be signaled on the next completed request.
4. Clients poll the completion queue to retrieve information on the completed work request.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

5. The completion information is returned to the user synchronously as a result of polling.

Specific requirements of the work-processing module are listed below:

- Completion signaling is controlled by the user
- Users specify whether completion notification is provided via callbacks or event signaling

4.3.3.2 Unreliable Datagram Services

A more detailed operation of a general unreliable datagram QP is outlined below.

Operation:

- The client creates a QP.
- The access layer returns a handle to the newly created QP.
- The client initializes the QP for datagram transfers.
- The client creates an address vector that references the remote destination.
- The access layer returns a handle to the requested address vector.
- The client submits a work request, indicating the QP, QKey, and address vector of the destination.
- The access layer posts the work request for the client.
- The access layer notifies the client of the send completion via the CQ specified when the QP was created.
- The client retrieves the send completion information from the associated CQ.

4.3.3.3 MAD Work Processing

Work processing is handled differently than that shown in Figure 4-12 for clients using the MAD services provided by the access layer. With the use of MAD services, the access layer processes work requests before passing them on to the queue pair. Likewise, the access layer performs post-processing on completion requests. Figure 4-13 highlights the differences.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

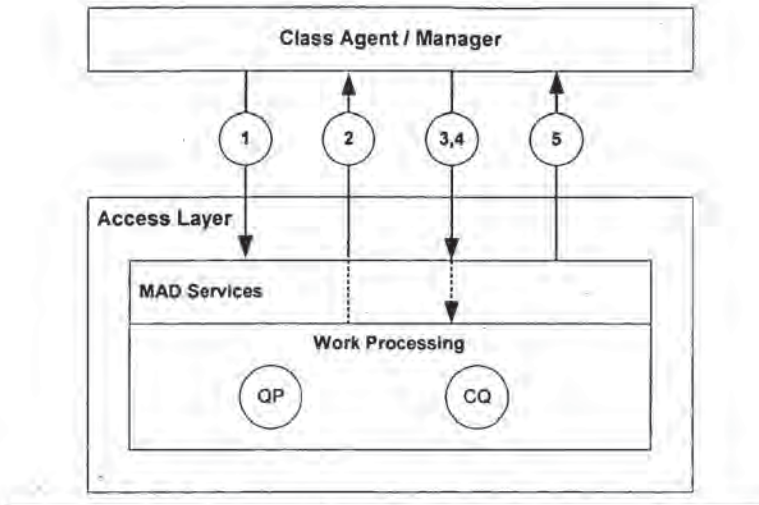


Figure 4-13: Work Processing with MAD Services

Operation:

1. Users indicate that a service should perform a specified task. The indication may come from the posting of a work request, the issuing of a MAD, or the invocation of a function call.
- 2-4. Steps 2-4 are similar to steps 2-4 in Figure 4-12. These steps only occur for users who request MAD services, but are using queue pair redirection. In such cases, when a user polls a completed work requests from a completion queue, the request is first given to the access layer for additional processing. The completed request is only returned to the user after all post-processing has completed.
5. The completion information is returned to the user asynchronously through a callback.

In general, most users of client specified services would only interface to steps 1 and 5 listed above.

4.3.4 Memory Manager

The memory manager provides users a way to register and deregister memory with a specified channel adapter. Memory must be registered before being used in a data transfer operation. Memory registration is considered separately from resource management for two main reasons. It is considered a speed path operation for kernel-mode users and uses synchronous deregistration. The memory registration process is shown in Figure 4-14.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

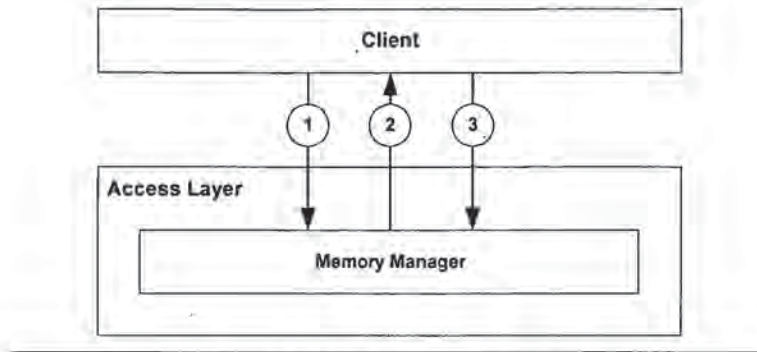


Figure 4-14: Memory Registration

Operation:

1. Clients register memory with the memory manager. Kernel-level users may register either physical or virtual memory. User-level clients cannot access to physical memory and may therefore only register virtual memory.
2. The memory manager returns a handle to the registered memory region, along with a key usable by a remote system to access the memory through an RDMA operation.
3. Once the memory is no longer being used to support data transfer operations, the client de-registers the memory region.

Requirements:

- Virtual and physical memory registration support
- Memory registration in the kernel is considered a speed-path operation

4.3.5 Operation Examples

4.3.5.1 Establishing a Connection to an IOC

The steps below provide a detailed example a client may follow to establish a connection to an IO controller.

Operation:

1. A client registers for notification of an IO controller with the Plug and Play manager. Refer to section 4.3.1.4 for information on the Plug and Play manager.
2. The access layer invokes a callback function to notify the client of the IO controller.
3. The client creates a protection domain.
4. The access layer returns a handle to the protection domain the client.
5. The client creates a CQ.
6. The access layer returns a handle to the CQ the client.
7. The client creates a QP.
8. The access layer returns a handle to the QP the client.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

9. The client obtains path records to the IO unit by querying the subnet administrator using information provided to the notification callback function. Refer to section 4.3.1.1.3.
10. The access layer returns the requested path records to the client.
11. The client formats a connection request structure providing the QP handle and selected path record and initiates the connection request process through the connection manager. Refer to section 4.3.1.2.
12. The access layer performs the connection request and invokes the client callback with the connection reply.
13. The client accepts the connection through the connection manager.
14. The access layer transfers a ready to use datagram to the remote node and returns to the client.
15. The client QP is now connected and ready for use.

4.3.5.2 Client Generated MAD Transaction

The steps below provide a detailed example a client may follow to perform a MAD transaction with a class management entity.

Operation:

1. The client obtains the service record information of the remote node by querying the subnet administrator via the access layer. Refer to section 4.3.1.1.3
2. The access layer returns the requested service record to the user.
3. The client obtains path records to the remote node by querying the subnet administrator using the information returned in the service record. Refer to section 4.3.1.1.3.
4. The access layer returns the requested path records to the client.
5. The client creates a QP. They may create their own QP or create an alias to one of the special QPs.
6. The access layer returns a handle to the newly created QP or the QP alias to the client.
7. The client initializes the QP for datagram transfers, specifying callback information for handling completed MAD work requests.
8. The client registers the QP with a management class to request MAD support from the access layer. A client can register a single QP with multiple management classes.
9. The client creates an address vector that references the remote destination.
10. The access layer returns a handle to the requested address vector.
11. The client formats a work request, indicating the QP, QKey, and address vector of the destination.
12. The access layer posts the work request for the user. If a response is expected, the access layer continues to resend the request until a response is received or the request times out.
13. The client is notified of the send completion through a send MAD callback.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

4.3.5.3 Joining a Multicast Group

The steps below provide a detailed example a client may follow to join and transfer datagrams with a multicast group.

Operation:

1. The client creates a QP.
2. The access layer returns a handle to the newly created QP.
3. The client initializes the QP for datagram transfers.
4. The client may post receive work requests to the QP.
5. The client obtains the PKey of the multicast group. The method used to obtain this PKey is implementation specific.
6. The client obtains the MGID of the multicast group. The method used to obtain the MGID is implementation specific.
7. The client joins the multicast group.
8. The access layer adds the QP as a group member, attaches the QP to multicast group, and returns the member record to the client. The client may receive datagrams from the multicast group before this call returns.
9. The client obtains path records to the group MGID by querying the subnet administrator using the information returned in the service record. Refer to section 4.3.1.1.3.
10. The access layer returns the requested path records to the client.
11. The client creates an address vector that references the MLID.
12. The access layer returns a handle to the requested address vector.
13. The client formats a work request, indicating the well-known QP number, and the QKey and address vector of the multicast group.
14. The access layer posts the work request for the client.
15. The client is notified of the send completion through a send callback.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

5. Architectural Specification – Subnet Management

5.1 Introduction

The OpenSM Subnet Manager infrastructure provides a facility for subnet discovery and activation. OpenSM also provides Subnet Administration capabilities to support clustering and IPC.

5.2 System Overview

The Subnet Management system resides in User Mode above the Access Layer interface, as depicted in the following diagram.

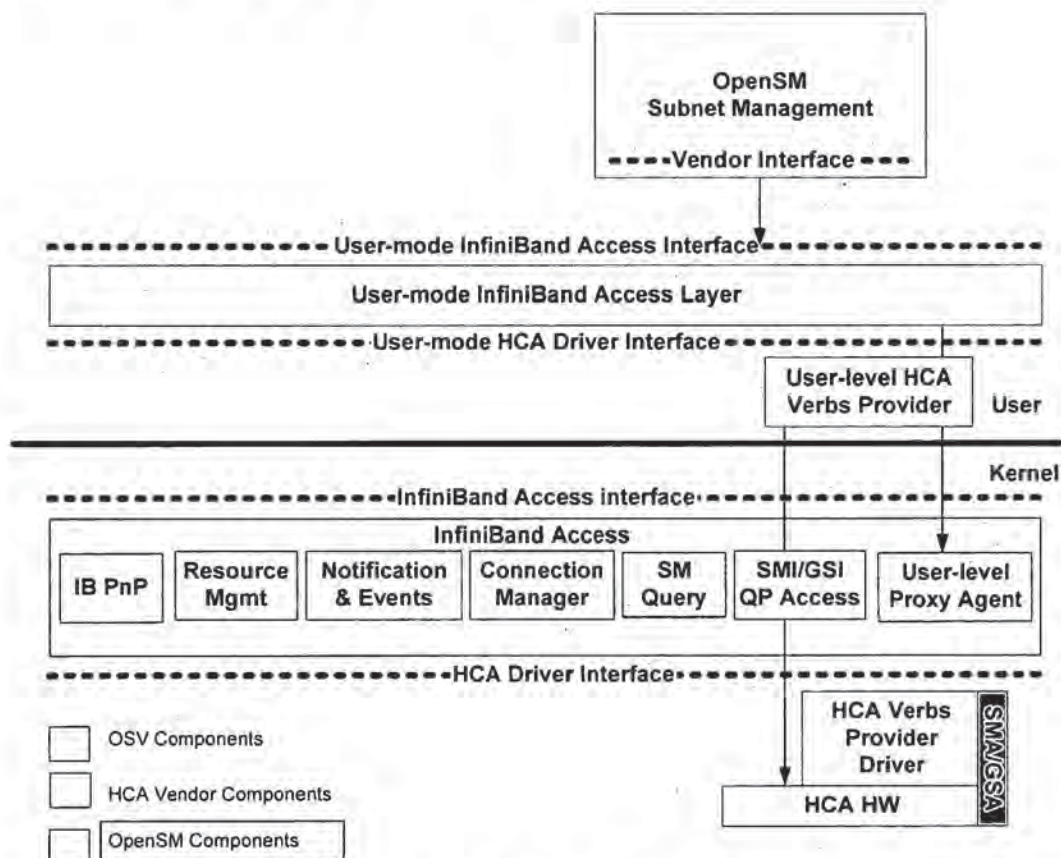


Figure 5-1: Subnet Management on the IB stack

As shown in Figure 5-1, OpenSM uses its internal Vendor Interface API's to send and receive QP0 and QP1 MAD packets. The Vendor Interface layer encapsulates and abstracts the underlying user mode InfiniBand Access API. This encapsulation allows developers to easily port OpenSM to other standard or proprietary InfiniBand access API's.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

Pictorial of Component Structure

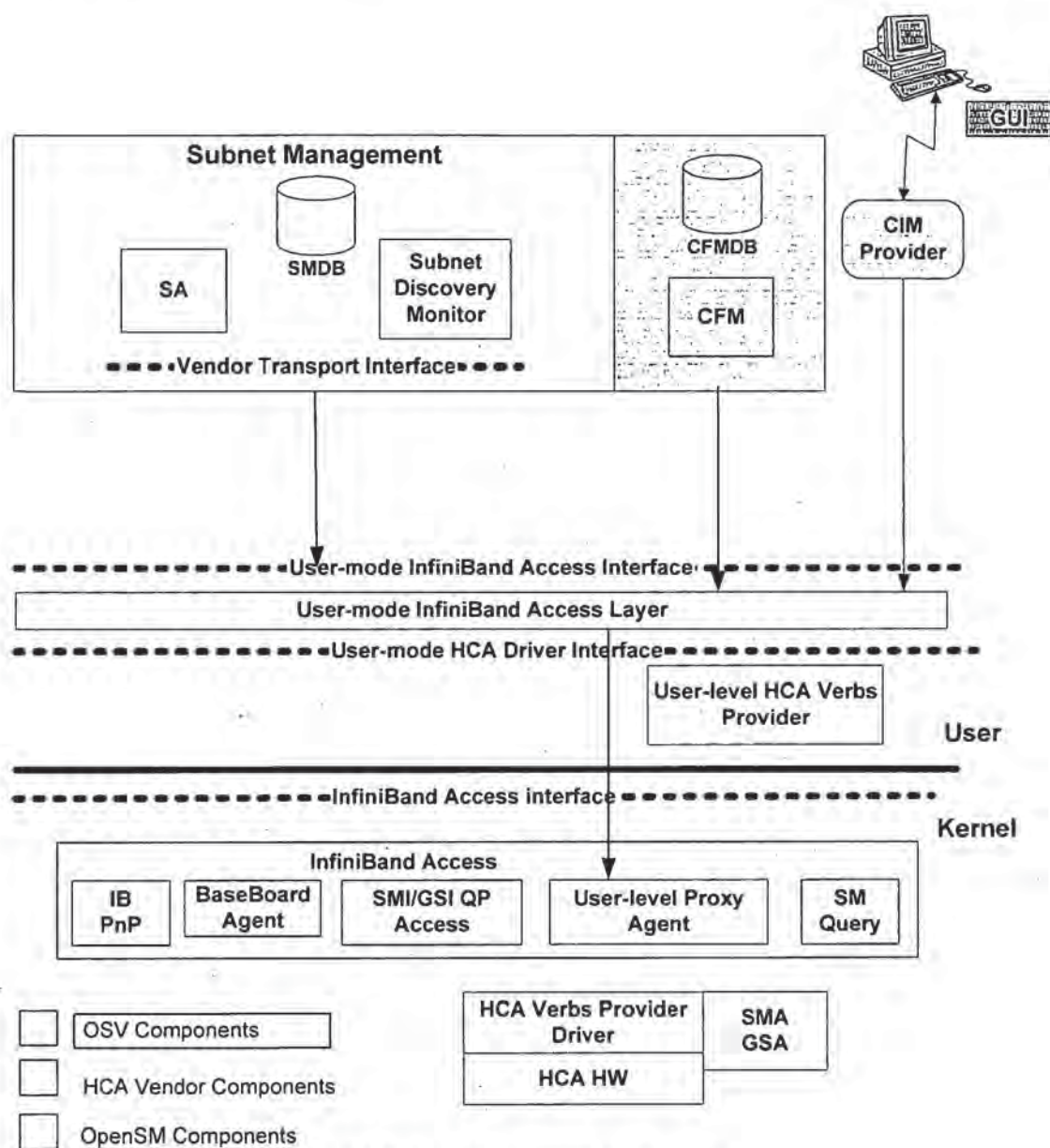


Figure 5-2: Subnet Management Components

As shown in Figure 5-2, the Subnet Management is composed of several components: Subnet Manager for discovery and configuration of the fabric, Subnet Manager Database (SMDB) for storing fabric data and Subnet Administration (SA) for handling IB queries against the SMDB, Three of the modules shown in Figure 5-2 are shaded to indicate these will not be available in the initial release of OpenSM: CFM, CIM Provider and a GUI (Graphical User Interface).

Rev. 1.0.1 - 01/Aug/02 01:48 PM

5.2.1 Relevant Specifications

The Subnet Manager conforms to the IBTA 1.0.a Specification as described in Chapters 13 and 14. The SA is described in Chapter 15.

5.2.2 List of External Interfaces

5.2.2.1 APIs

API Name and version	Brief Description
User mode InfiniBand Access Layer	See Chapter 4

5.2.3 List of Product Deliverables

5.2.3.1 User Interfaces

SM CLI	Normally the SM will start as a system daemon, but there is an optional interface to launch the SM from the command line.
--------	---

5.2.3.2 Product Documentation

SM User's Guide	Documents the use of the SM
SM man page	Linux style man page for the SM CLI.
Installation Guide	Describes the installation and un-installation of the SM components

5.3 Theory of Operation

5.3.1 Initialization

The SM normally starts as a Linux daemon, started from the `/etc/rc` scripts. It may be optionally started by the system administrator from a shell.

5.3.2 SM Functionality and Interfaces

The Subnet Manager is responsible for the configuration of the subnet. On startup, the SM discovers the nodes of the fabric using directed-route SMP packets sent through the User mode InfiniBand Access Layer (uAL). The SM assigns LIDs to the switches and channel adapters, sets up the switch forwarding tables, and activates the ports on the subnet. It then continues to periodically sweep the fabric to identify new or newly disconnected links. If the SM discovers another SM on the same subnet, it will use the IBTA specified negotiation for the SM Master role. It is expected that only in the case of losing the negotiation to an SM from the same vendor that the fabric state (LID assignments, port state, forwarding table state) will be seamlessly maintained.

The SM maintains fabric data in the Subnet Management Database (SMDB), allowing the Subnet Administration (SA) to respond to SA record queries for subnet information. Generally speaking, the SM writes to SMDB and the SA reads from the SMDB.

The SM plays a relatively small role in automatic path migration. When connections are established, the connecting parties may request multiple Path Records from the SA, and select

Rev. 1.0.1 - 01/Aug/02 01:48 PM

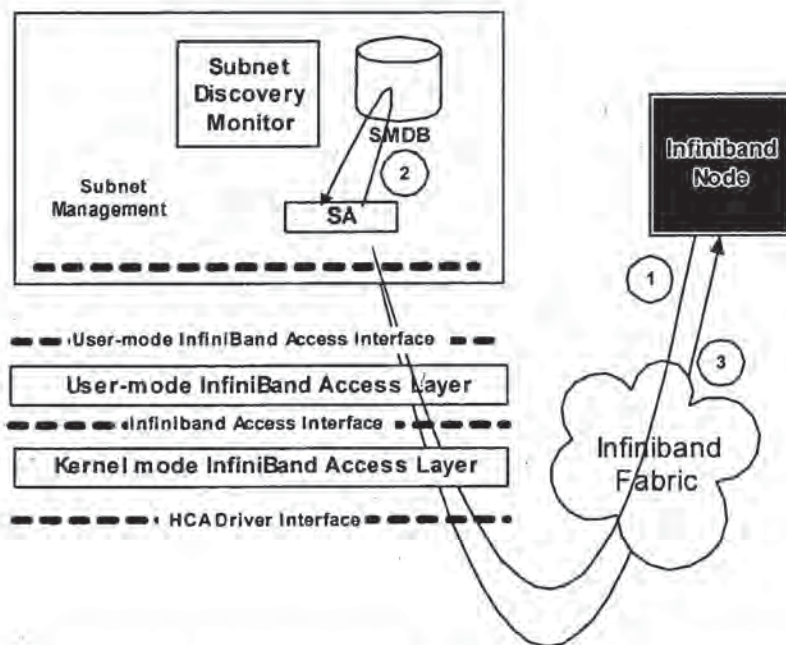
one path (DGID/SGID pair) to be primary and the other to be the alternate path. For this to work properly, the SM must configure ports with LMC greater than zero, such that a port may have multiple LIDs. The switch forwarding tables must be configured so that the various paths have as little overlap as possible, given the configuration of the links on the fabric. The SA must be able to return multiple path records when these conditions exist. The SM is not involved in the actual migration of the path at the time of link failure.

The SM is responsible for managing the Multicast Forwarding Tables of the switches in the subnet. The SM is informed of changes to the multicast configuration via local communication with the SA, which in turn handles the SA MCMemberRecord messages from end nodes requesting to join or leave a multicast group.

5.3.3 SA Functionality

Through the use of Subnet Administration class MADs, SA provides access to and storage of information of several types as defined in IBTA Spec 1.1 Chapter 15. For the initial release of OpenSM, SA will support only the required information (SA Attributes) that is needed to support SRP (chapter 6), IPoIB (chapter 7) and SDP (chapter 8) channel drivers. SA attributes supported are: NodeRecord, PortInfoRecord, PathRecord, MCMemberRecord and ServiceRecord.

5.3.3.1 SA Query Data Path



All nodes on the fabric obtain all the fabric information, such as Node, Port or Path Records, by sending standard SA Record queries (1) to the SA, using the User Access Interface, which retrieves the appropriate data (2) from the SMDB, and returns the Query record (3) through the SM Vendor Transport Interface via the OSV Access Layer.

5.3.4 OpenSM Internals in Brief

OpenSM is based on a dispatcher and controller model. The dispatcher is a pool of threads that handle dispatching of messages between the controllers. Each controller handles a narrowly

Rev. 1.0.1 - 01/Aug/02 01:48 PM

scoped portion of the workload. For example, one such controller is the MAD controller that communicates with the vendor transport layer to receive MADs. The MAD controller parses the MAD to determine which other controller should perform the needed processing. The MAD controller then posts this MAD to the dispatcher for processing by the recipient controller as soon as a worker thread becomes available.

This modular approach allows developers to quickly add modules (controllers) that plug into the framework to support new features. Typically the number of worker threads in the dispatcher equals the number of CPUs in the system. All controllers run in the context of the dispatcher thread that invokes them. External components such as the SMDB are shared by all the controllers. Refer to OpenSM HLD/LLD for details on this design.

5.3.5 OpenSM Multi-threaded Discovery

OpenSM performs fully multi-threaded subnet sweeps (initial subnet discovery is simply the first sweep). One implication is that during any subnet sweep, OpenSM may discover switches and nodes in a different order than during a previous sweep. OpenSM is designed such that the order in which it discovers subnet objects is irrelevant to both the internal representation of the subnet in the SMDB and the final configuration assigned to the subnet itself.

5.3.6 OpenSM Non-disruptive Subnet Configuration

To minimize traffic disruption, OpenSM attempts to preserve the existing configuration of a subnet. This situation can occur when OpenSM inherits a subnet due to failure of an established SM, when the OpenSM node itself is rebooted in an active subnet, or when two or more active subnets merge. Where possible OpenSM preserves existing LID assignments. LID conflicts are resolved by reassigning LID values to one of the conflicting nodes.

5.3.7 LMC > 0

OpenSM supports Lid Mask Count (LMC) values greater than 0, which allows multiple paths between endpoints on a subnet. Specifically, the number paths permitted is 2^{LMC} . Given equal hop count paths, OpenSM attempts to minimize path overlap when configuring switch forwarding tables. OpenSM will not configure a paths with less overlap if those paths contain more hops than an overlapping path. Future implementations may consider other factors in redundant path selection such as link width and the number of VLs supported by alternative paths.

The LMC value selected is a user configuration parameter.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

6. Architectural Specification – SRP Driver

6.1 Introduction

Host systems require access to remote storage devices across an InfiniBand fabric. The method used to access these devices is defined by the IO protocol. The SCSI RDMA Protocol (SRP) developed by the ANSI INCITS T10 Technical Committee is designed to take full advantage of the features provided by the InfiniBand Architecture. In addition, the SCSI command set is widely used throughout the industry and is applicable to a wide variety of device types. SRP allows a large body of SCSI software to be readily used on InfiniBand Architecture and is rapidly emerging as the protocol of choice for block-based storage. This section describes the architecture of the Linux SRP device driver.

6.2 Overview

User level processes typically access storage devices through a file system. The Linux operating system supports a number of file systems. The range of file systems supported is made possible through a unified interface to the Linux kernel known as the Virtual File System (VFS). The VFS interface provides a clearly defined interface between the kernel and the different file systems. VFS maintains internal structures, performs standard actions, and forwards tasks to the appropriate file system driver.

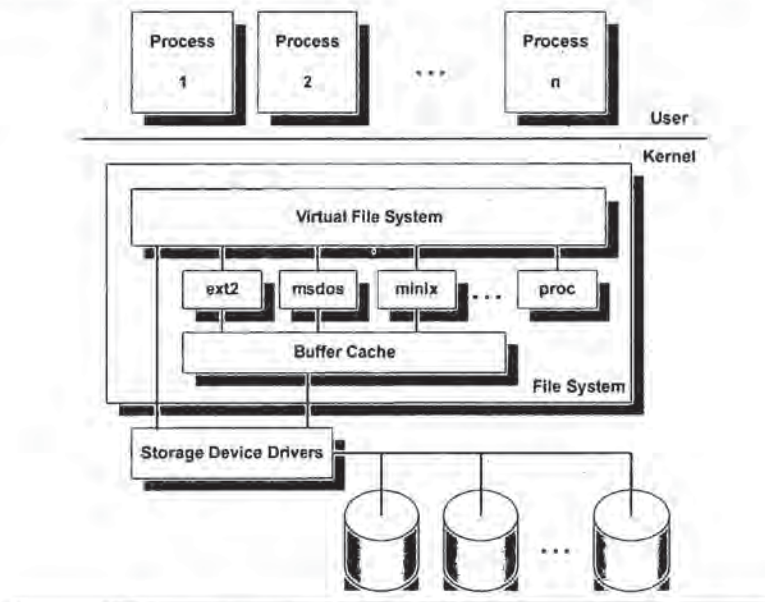


Figure 6-1: Linux File System Components

The central demand made of a file system is the purposeful structuring of data to allow high performance and randomized access. Linux employs a dynamic buffer cache to increase the performance of block devices accessed through the file system. In addition to storage device access through a file system, a process may also access a raw or character storage device directly, bypassing the file system and buffer cache components.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

The storage device drivers provide access to physical devices by abstracting the details of the underlying hardware interface, for example IDE or SCSI. The SCSI device driver stack provides access to devices supporting SCSI protocols. The Linux SCSI framework contains an abstraction layer called the SCSI mid-layer. This layer provides upper-level drivers and applications with a device-independent set of interfaces to access devices. The SCSI mid-layer routes IO requests to low-level SCSI Host Bus Adapter (HBA) specific drivers. Low-level SCSI drivers provide access to devices through particular HBA hardware interfaces. Multiple low-level SCSI drivers may be loaded at the same time as required by the underlying hardware. This model simplifies the implementation of both the hardware-specific HBA drivers (the low-level SCSI drivers) and applications. The SRP device driver is implemented as a Linux low-level SCSI device driver.

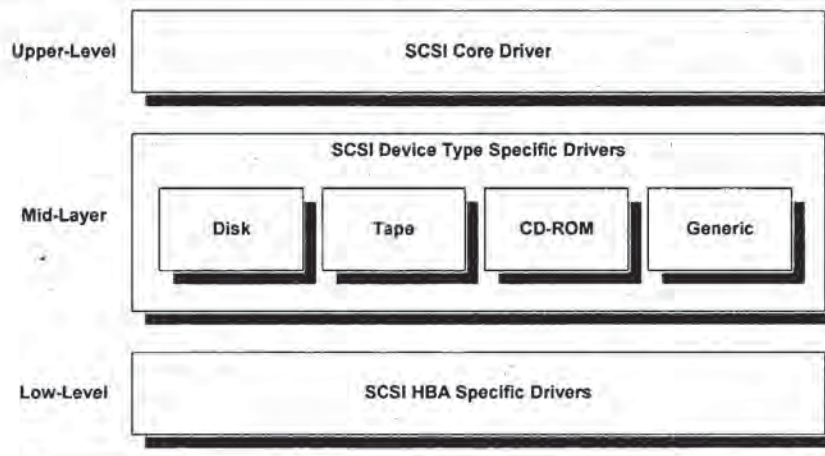


Figure 6-2: Linux SCSI Driver Stack

The SRP device driver differs from traditional low-level SCSI drivers in Linux in that the SRP driver does not control a local HBA. Instead, it controls a connection to an IO controller to provide access to remote storage devices across an InfiniBand fabric. The IO controller resides in an IO unit and provides storage services.

The SRP device driver uses the interface provided by the InfiniBand Access Layer to communicate across an InfiniBand fabric to an IO controller. Thus, the SRP device driver is not dependent on any particular channel adapter implementation.

Rev. 1.0.1 - 01/Aug/02 01:48 PM

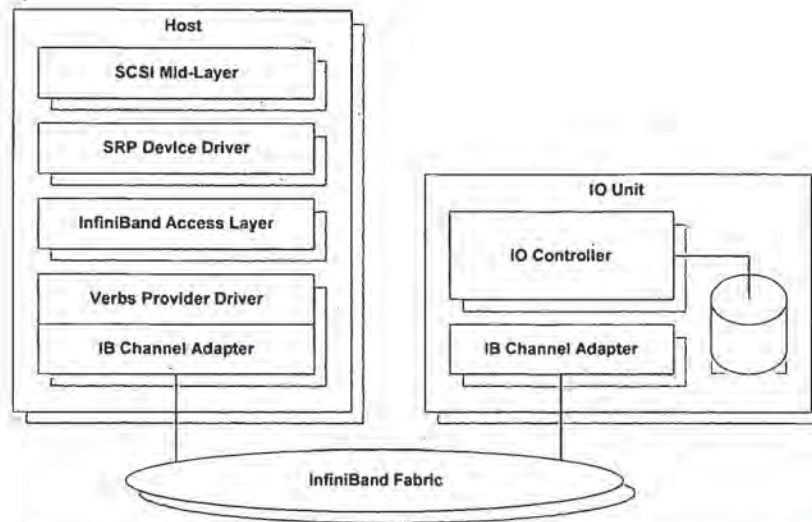


Figure 6-3: SRP Device Driver / IO Controller Relationship

The SRP device driver is known as the SRP initiator whereas the service on the IO controller is known as the SRP target. The SRP Specification defines the communication protocol used between the initiator and the target.

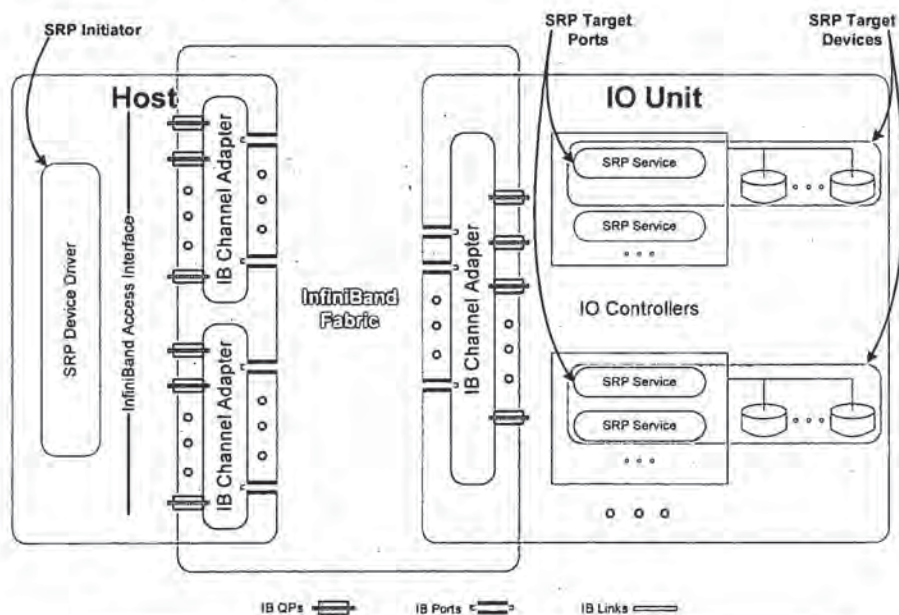


Figure 6-4: SRP Architecture Mapping

The SRP protocol provides transport services to enable a basic client-server model where an initiator presents SCSI tasks to a target for execution. All operations in this model use reliable service connections across the InfiniBand fabric. SRP defines the message format and behavior

Rev. 1.0.1 - 01/Aug/02 01:48 PM

required to transfer commands and data between an initiator and a target. SCSI commands and completion status are exchanged asynchronously using message send operations.

SRP defines a buffer management mechanism that allows a target to limit the number of requests that may be queued on the target for execution. A target may use this mechanism to manage internal resources, for example, to dynamically allocate message buffers among multiple initiators. This allows the target to provide optimal use of limited resources and improve overall system performance.

In SRP, the target performs all device data transfers to or from initiator memory using RDMA operations. An initiator allows RDMA access from the target by registering its data buffer memory. Information describing the registered data buffer memory is included in the SRP command.

A typical SRP IO transaction is as follows:

1. The initiator builds an SRP request message that contains a SCSI command, a device logical unit number, and a data buffer memory descriptor, and sends the request to the target.
2. The target receives the SRP request and performs an RDMA operation to transfer the initiator data buffer memory contents to or from the device.
3. The target builds an SRP response message indicating the completion status of the request and sends the response to the initiator.

The initiator can also perform SRP task management operations, for example, to abort a task on the target. In addition, the target can send messages to the initiator describing asynchronous events, such as the insertion of new media into a removable device.

6.3 Theory of Operation

The following subsections describe the operation of the Linux SRP device driver.

6.3.1 Load / Unload

The InfiniBand Access Layer provides a Plug and Play manager. The Plug and Play manager maintains a mapping of IO controller profile class, subclass, and protocol identifier values to channel drivers. If a channel driver is not present at the time the bus driver needs to notify it of a new IO controller, then the Plug and Play manager will load the channel driver. The Plug and Play manager delivers add and remove notifications to a channel driver asynchronously through a function call interface registered with the bus driver when the channel driver is loaded.

Rev. 1.0.1 - 01/Aug/02 01:48 PM