

Enhancing Linux Scheduler Scalability

Mike Kravetz
IBM Linux Technology Center

Hubertus Franke, Shailabh Nagar, Rajan Ravindran
IBM Thomas J. Watson Research Center
{mkravetz,frankeh,nagar,rajancr}@us.ibm.com
<http://lse.sourceforge.net>

Abstract

This paper examines the scalability of the Linux 2.4.x scheduler as the load and number of CPUs increases. We show that the current scheduler design involving a single runqueue and lock can suffer from lock contention problems which limits its scalability. We present alternate designs using multiple runqueues and priority levels that can reduce lock contention while maintaining the same functional behavior as the current scheduler. These implementations demonstrate better overall scheduling performance over a wide spectrum of loads and system configurations.

1 Introduction

Linux has seen tremendous growth as a server operating system and has been successfully deployed in enterprise environments for Web, file and print serving. Often, the increased demand in such environments can be met by horizontally scaling the system with clustering. For such applications, the operating system needs to efficiently support SMPs consisting of only a small number of CPUs.

More demanding applications, such as database, e-business or departmental servers, tend to be deployed on larger SMP systems. To support such applications, Linux must scale well *vertically* as more CPUs are added to an SMP. It must also scale with the increased number of processes and threads that such SMPs are expected to handle. In both these situations, the scheduler can be a key factor in achieving or limiting operating system scalabil-

ity. The current Linux scheduler (2.4.x kernel) has two defining characteristics. First, there is a *single* unordered runqueue for all runnable tasks in the system, protected by a single spinlock. Second, during scheduling, *every* task on the runqueue is examined while the runqueue lock is held. These have a two-fold effect on scalability. As the number of CPUs increases, there is more potential for *lock contention*. As the number of runnable tasks increases, *lock hold* time increases due to the linear examination of the runqueue. Independent of the number of CPUs, increased lock hold time can also cause increased lock contention, depending on the frequency of scheduling decisions. For spinlocks, increased lock hold time and lock contention result in a direct increase in lock wait time which is a waste of CPU cycles. These observations are reinforced by recent studies. Measurements using Java benchmarks [2] show that the scheduler can consume up to 25% of the total system time for workloads with a large number of tasks. Another study [3] has observed run queue lock contention to be as high as 75% on a 32-way SMP.

Lock contention problems can generally be addressed in two ways. First, the protected data structure can be reorganized so that it can be traversed faster with a corresponding decrease in the average lock hold time. Second, the data structure can be broken up or partitioned into smaller parts, each protected by its own separate lock. This reduces the probability of lock contention overall. Additionally, it allows multiple examinations of the subparts to proceed in parallel, reducing lock wait time for the data structure as a whole.

The main contribution of this paper is the design, implementation and evaluation of two new Linux schedulers which improve scalability using these two

approaches. The priority level scheduler (PLS) aims at reducing lock hold time by maintaining runnable tasks in priority lists. The multiqueue scheduler (MQ) reduces lock contention by maintaining per-cpu runqueues. Both of these solutions are deployed on commercial operating systems but have not been seriously considered for Linux. Though priority level schedulers have been implemented for Linux [5] and have shown improvements over the vanilla scheduler, we show here that the reduction in lock hold time by such methods only improves scalability with an increased number of tasks. However, it is not sufficient to improve scalability with increasing CPU counts. In particular, though our PLS also does better than the current scheduler at moderate to high task counts, MQ outperforms the current scheduler *and* PLS over a wide range of workloads. More importantly, these improvements are obtained while maintaining functional equivalence with the current scheduler, leaving room for further improvements.

The rest of the paper is organized as follows. Section 2 presents a description of the implementation of the current scheduler. The parts which define the functionality (and need to be retained) are identified along with the bottlenecks. Section 3 presents the priority queue scheduler implementation. The main contribution of this paper, the multiqueue scheduler, is described in Section 4. Results using microbenchmarks and a decision support workload are shown in Section ???. Section 5 concludes with directions for future work.

2 Default SMP Scheduler (DSS)

The default SMP scheduler (DSS) in Linux 2.4.x treats processes and threads the same way, referring to them as tasks. Each task has a corresponding data structure which maintains state related to address space, memory management, signal management, open files and privileges. Traditional threading models and light-weight processes are supported through the `clone` system call.

For the purpose of scheduling, time is measured in architecture-dependent units called ticks. On x86 systems, timer ticks are generated at a 10ms resolution. Each task maintains a counter (`tsk->counter`) which expresses the time quantum for which it can execute before it can be preempted.

By decrementing this counter on timer tick interrupts, DSS implements a priority-decay mechanism for non-realtime tasks. The priority of a task is determined by a `goodness()` value that depends on its remaining time quantum, `nice` value and the affinity towards the last CPU on which it ran. DSS supports preemption of tasks only when they run in user mode. The responsiveness of lengthy kernel code can be increased by checking for scheduling requirements at appropriate locations. Priority preemption can occur any time the scheduler runs.

The kernel scheduler consists of two primary functions :

1. `schedule(void)` : This function is called synchronously by a processor to select the next task to run e.g. at the end of `sleep()`, `wait_for_IO()` or `schedule_timeout()`. It is also called preemptively on the return path from an interrupt e.g. a reschedule-IPI (interprocessor interrupt) from another processor, I/O completion or system call. In such cases, the `schedule()` function is called if the `need_resched` field of the current task is set.
2. `reschedule_idle(task_struct *tsk)` : This function is called in `wake_up_process()` to find a suitable processor on which the parameter task can be dispatched. `wake_up_process()` is called when a task is first created or when it has to be re-entered into the runqueue after an I/O or sleep operation. `reschedule_idle()` tries to find either an idle processor or one which is running a task with a lower goodness value. If successful, it sends an IPI to the target CPU, forcing it to invoke `schedule()` and preempt its currently running task.

Internally, the scheduler maintains a single runqueue protected by a spinlock. The queue is unordered, which allows tasks to be inserted and deleted efficiently. However, in order to select a new task to run, the scheduler has to lock and traverse the entire runqueue, comparing the goodness value of each schedulable task. A task is considered schedulable if it is not already running and it is enabled for dispatch on the target CPU. The goodness value, determined by the `goodness()` function, distinguishes between three types of tasks : realtime tasks (values 1000+), regular tasks (values between 0 and 1000) and tasks which have yielded the processor (value -1). For regular tasks, the goodness value

consists of a static or non-affinity part and a dynamic or affinity part. The non-affinity goodness depends on the task's `counter` and `nice` values. The affinity part accounts for the anticipated overheads of cache misses and page table switches incurred as a result of migrating tasks across CPUs. If the invoking CPU is the same as the one the task last ran on, the goodness value is boosted by an architecture dependent value called `PROC_CHANGE_PENALTY`. If the memory management object (`tsk->mm`) is the same, goodness values are boosted by 1. The counter values of all tasks are recalculated when all schedulable tasks on the runqueue have expired their time quanta. Due to space limitations, we refer the reader to detailed descriptions of DSS in [5, 1].

3 Priority Level Scheduler (PLS)

The priority level scheduler (PLS) seeks to reduce the number of tasks examined during a scheduling decision. It reorganizes the single runqueue of the default SMP scheduler (DSS) into an array of lists indexed by the non-affinity goodness of tasks. The indices of the currently running task, and the highest schedulable task together with an affinity boost, determine the range of lists to be searched for the next candidate. The priority lists are still protected by a single runqueue lock as they conceptually provide a single runqueue.

In our implementation, we coalesce all realtime tasks into a single list at the highest index. This method of enqueueing tasks results in 61 lists for the x86 platform and up to 335 lists for other architectures. A task's goodness value can change during its execution, e.g. during `fork`, `timer`, `exit`, and `recalculate`, requiring it to be reassigned to a different priority list. To avoid frequent requeueing, yielding tasks are enqueued according to their non-yield goodness values and handled appropriately while walking the lists. The implementation ensures that yielding tasks do not execute before any other runnable task.

At `schedule()` time, the currently running task is the default candidate to run next, and if it is not yielding, also establishes the lowest list to be scanned (as no task on a lower list can receive an affinity boost which results in a priority higher than that of the currently running task.) If the task stopped executing, e.g. due to I/O wait, the

`idle-task` becomes the default candidate and all lists need to be searched by default. Tasks with expired counters fall into the lowest list and are never inspected.

The determined range of lists is now scanned in top-down priority order for non-yielding schedulable tasks and if one is found and its goodness value is better than the default candidates, it becomes the default candidate. Further search can be limited to lists whose priority lie within `PROC_CHANGE_PENALTY` of the default candidate's list, as no list below that can have a higher goodness value even after getting an affinity boost. Even within this range, the search can be terminated as soon as a task is found that last ran on the invoking CPU. As a further optimization, we maintain a bitmap of non-empty list indices that allows us to efficiently skip empty lists, using the `find_first_zero()` function. We disregard the `tsk->mm` boost, which essentially provides a tie-breaker between two task of equal priority, as it would require a complete scan of the last reached list and the one below it. We have also implemented versions of priority level schedulers that account for the `tsk->mm` boost but only observed infrequent differences in scheduling behavior compared to DSS, while suffering from degraded performance. We chose to present the best performing PLS implementation to highlight the need for reducing lock contention as done in MQ. We have also implemented versions that limit the number of lists, by utilizing a different hash function, but did not observe performance improvements. Since PLS keeps running tasks on the runqueue (i.e. in their list) and therefore inspects these tasks during scheduling, we can expect that for low task counts (\approx #CPUs), PLS will introduce additional overhead compared to DSS. However, with the increase in the number of tasks, the probability of finding a task that ran last on the invoking CPU increases as does the benefit of limiting the number of tasks that need to be traversed. Together, we expect an reduced average lock hold time.

4 Multi-Queue Scheduler (MQ)

The multi-queue scheduler (MQ) is designed to address scalability by reducing lock contention and lock hold times while maintaining functional equivalence with DSS. It breaks up the global run-queue and global run-queue lock into corresponding per-

CPU structures. Lock hold times are reduced by limiting the examination of tasks to those on the runqueue of the invoking CPU along with an intelligent examination of data corresponding to the non-local runqueues. Moreover, the absence of a global lock allows multiple instances of the scheduler to be run in parallel, reducing lock wait time related to lock contention. Together these reduce the scheduler related lock contention seen by the system.

MQ defines per-CPU runqueues which are similar to the global runqueue of the DSS scheduler. Related information such as the number of runnable tasks on this runqueue is maintained and protected by a per-CPU runqueue lock.

The `schedule()` routine of MQ operates in two distinct phases. In the first phase, it examines the local runqueue of the invoking CPU and finds the best local task to run next. Schedulers incorporating only this phase exist [4], but can lead to problems of priority inversion and load imbalances amongst the runqueues. The load imbalance problem is illustrated in Fig 1 which shows the deviations from the mean runqueue length over time for 4-way SMP executing a kernel build and using such a restricted multi-queue scheduler. MQ directly addresses priority inversion in the second phase by comparing the local candidate with the top candidates from remote runqueues before making the final selection. This also has a load balancing effect.

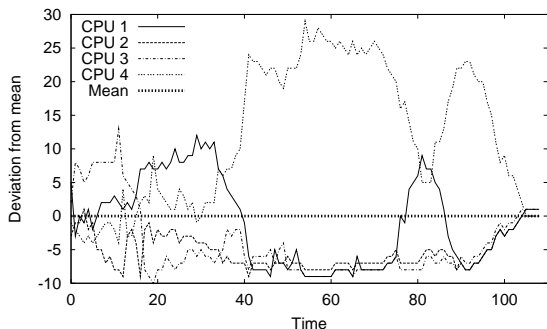


Figure 1: Deviation from mean of runqueue lengths for a 4-way SMP during a kernel build and running a scheduler which only looks at the local runqueue

In more detail, the `schedule()` routine of MQ acquires the runqueue lock of the invoking CPU's runqueue and scans the latter looking for the schedulable task with the highest goodness value. To facilitate the global decision in the second phase, it also records the second highest non-affinity good-

ness value in the `max_na_goodness` field of the local runqueue. The non-affinity goodness (henceforth called `na_goodness`) is the goodness value of a task without any consideration for CPU or memory map affinity. The best local candidate's goodness value (which includes appropriate affinity boosts) is compared with the `max_na_goodness` of all other runqueues to determine the best global candidate. If the global candidate is on a remote runqueue, `schedule()` tries to acquire the corresponding lock and move the candidate task over to its local runqueue. If it fails to acquire the lock or the remote task is no longer a candidate (its `na_goodness` value has changed), `schedule()` skips the corresponding runqueue and tries again with the next best global candidate. In these situations, MQ's decisions deviate slightly from those made by DSS e.g. the third best task of the skipped runqueue could also have been a candidate but is not considered as one by MQ.

The `reschedule_idle()` function attempts to find a CPU for a task which becomes runnable. It creates a list of candidate CPUs and the `na_goodness` values of tasks currently running on those CPUs. It chooses a target CPU in much the same way as the `schedule()` routine, trying to acquire a runqueue lock and verifying that the `na_goodness` value is still valid. Once a target CPU is determined, it moves the task denoted by its argument onto the target CPU's runqueue and sends an IPI to the target CPU to force a `schedule()`. `reschedule_idle()` maintains functional equivalence with DSS in other ways too. If a task's previous CPU is idle, it is chosen as the target. Amongst other idle CPUs, the one which has been idle the longest is chosen first.

MQ's treatment of realtime tasks takes into account the conflicting requirements of efficient dispatch and the need to support Round Robin and FIFO scheduling policies. Like DSS, it keeps runnable realtime tasks on a separate global runqueue and processes them the same way.

An important aspect of MQ's implementation is the care taken to avoid unnecessary cache misses and false sharing. Runqueue data is allocated in per-CPU cache-aligned data structures.

5 Conclusion and Future Work

The Linux 2.4 kernel provides a concise SMP scheduler that does well for small SMPs running moderate loads. However, we have shown that, as the number of CPUs or the load increases, the scalability limitations of the scheduler start showing up. Profiling data for a range of workloads show that the problem is due to high lock contention and large lock hold times.

Reducing the lock hold times, as is done in the PLS scheduler presented here, does alleviate the problem somewhat with a corresponding improvement in scalability. However, this is not sufficient to address the overall scalability as the number of CPUs increases. Also, at low loads, the overheads of PLS make it perform worse than DSS. The MQ scheduler directly addresses lock contention by breaking up the single runqueue and its associated locks into per-CPU equivalents. This brings a significant improvement in lock contention, scalability and overall performance of the scheduler.

We are currently working on more extensive evaluations of the ideas presented in this paper. We want to use more realistic workloads such as those seen on compute and database servers. Further extending the MQ design, we are looking at schedulers which use *CPU pooling*. CPU pooling divides the CPUs of a system into a set of pools. Each pool consists of one or more CPUs. Scheduling decisions are localized to the individual CPU pools, and load balancing algorithms are put in place to balance the load among the pools. CPU Pooling provides a continuum between complete runqueue separation, as provided in [4], and MQ with its global scheduling decisions.

It is our belief that CPU pooling will be beneficial on large SMP machines where making global scheduling decisions will become more expensive. In addition, CPU pooling may be a good choice for NUMA architectures where CPUs on individual compute nodes can be mapped to CPU pools.

6 Acknowledgments

We would like to thank the many people on the `lse-tech@lists.sourceforge.net` mailing list

who provided us with valuable comments and suggestions during the development of these alternative scheduler implementations. In particular, we would like to recognize John Hawkes, for running our implementations on some large systems at SGI, and Bill Hartner for related discussions and help with the experiments. This work was developed as part of the Linux Scalability Effort on SourceForge (`lse.sourceforge.net`). Here you can find more detailed descriptions of our scheduler implementations as well as the latest source code.

References

- [1] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Associates.
- [2] R. Bryant and B. Hartner. *Java Technology, Threads, and Scheduling in Linux*. *Java Technology Update*, 4(1), Jan 2000.
- [3] R. Bryant and J. Hawkes. *Lockmeter: Highly-Informative Instrumentation for Spin Locks in the Linux Kernel*. In *Proc. Fourth Annual Linux Showcase and Conference, Atlanta*, Oct 2000.
- [4] Hewlett Packard Inc. *Process resource managers for Linux : Linux plug-in schedulers*. <http://resourcemanagement.unixsolutions.hp.com/WaRM/schedpolicy.html>.
- [5] S. Molloy and P. Honeyman. *Scalable Linux Scheduling*. In *Usenix Annual Technical Conference (Freenix Track)*, June 2001. To appear.