

DLM  
TK  
7895  
.M4  
H35  
1978

# THE CACHE MEMORY BOOK

SECOND EDITION

JIM HANDY

DeLaMare Library

NOV 20 1998

Univ. of Nev. - Reno



Academic Press

San Diego New York Boston

London Sydney Tokyo Toronto

This book is printed on acid-free paper. ∞  
Copyright © 1998, 1993 by Academic Press, Inc.  
All rights reserved.  
No part of this publication may be reproduced or  
transmitted in any form or by any means,  
electronic or mechanical, including photocopy, recording, or  
any information storage and retrieval system, without  
permission in writing from the publisher.

ACADEMIC PRESS, INC.  
1250 Sixth Avenue, San Diego, CA 92101-4311

United Kingdom Edition published by  
ACADEMIC PRESS LIMITED  
24-28 Oval Road, London NW1 7DX

Library of Congress Cataloging-in-Publication Data

Handy, Jim.

The cache memory book / Jim Handy.—2nd ed.

p. cm.

Includes index.

ISBN 0-12-322980-4 (alk. paper)

1. Cache memory. I. Title.

TK7895.M4H35 1998

004.5'3—dc21

97-35868  
CIP

UNIVERSITY LIBRARY  
UNIVERSITY OF NEVADA, RENO  
RENO, NV 89557

Printed in the United States of America

97 98 99 00 01 EB 9 8 7 6 5 4 3 2 1

## CHAPTER 4

# MAINTAINING COHERENCY IN CACHED SYSTEMS

A statistic that is often quoted decrees that the most common cause of quarrels between married people is money. Why would such a statistic creep into a book about cache memory design? Because the parallels between some kinds of money arguments and the issue of cache coherency are both profound and down-to-earth.

Let's assume that the majority of the money arguments start with something like one of these examples:

"Why don't you enter it on the check register when you withdraw money from the automatic teller machine? Now you made us bounce three checks!"

"You charged \$900.00 onto the Visa account and *forgot* to tell me? No wonder my credit was declined at the restaurant in front of my boss!"

"That was the money we were saving to go to Hawaii!"

"So what *did* happen to that \$100 cash in your wallet, anyway?"

In all of these cases, there is money (or credit) which is community property to both members of the marriage, which one party spends without telling the other. If we move from married couples over to computers, and we change the community property over to the memory space, we can see that there are potential catastrophes if the same data is being manipulated by two different devices without either somehow informing the other of its actions. Each member of the couple thinks she or he has a mental picture of what is in the bank or what has been charged on the credit card, just like a cache memory is supposed to contain an accurate copy of the appropriate contents of main memory. When one or another of the couple alters what is in the main memory, then the other must be informed. Likewise, whenever any device on the system bus updates either main memory or a cache location which is to be copied to main memory at a later time, it must assure that no other device can harbor a misunderstanding about the freshness of the data in the main memory.

Such problems have been solved in a number of software databases. Imagine the problem of booking airline seats. United Airlines has a data base which can be accessed simultaneously by about 2,100 telephone ticket agents at each of three offices in the United States alone. Likewise, hundreds of travel agents in the United States are on-line and also have the power to book a seat on a flight. In a worst-case scenario, there could be a single seat remaining on a flight from New York to Chicago, and, at the exact same time, everybody who was on the system tried to sell that seat to a customer. If the problem were not solved, about 3,000 people might be assigned the same seat on the same airplane!

**Cache coherency** is the term given to the problem of assuring that the contents of the cache memory and those of main memory for *all* caches in a multiple cache system are either identical or under tight enough control that stale and current data are not confused with each other. Like any other cache buzzword, there are less-often-used alternatives to the word coherency, namely, **consistency** and **currency**. The term **stale data** is used to describe data locations which no longer reflect the current value of the memory location they once represented. As I type this book, the new version of this chapter is stored in the computer's main memory, while a stale version resides on the computer's hard disk. The next time I save the file, the disk and the main memory will contain the same data at the same locations and will be coherent with each other until I again start to type.

## 4.1 SINGLE-PROCESSOR SYSTEMS

At first flush, most designers assume that coherency is only a problem in multiprocessor systems, or possibly in systems with copy-back caches. It

seems as if a write-through cache in a single-CPU system would never have coherency problems. This is untrue due to any activities which are not under control of the processor, in other words, input and output activities.

The simplest example is that of a memory-mapped polled I/O device, which the processor is continually reading to determine the status of a single bit. We used the same example to illustrate the use of noncacheable areas in Section 2.2.7. If the cache contains a copy of the memory-mapped I/O location, and the CPU refers to the cached copy rather than to the I/O device, the processor will never see any change in the I/O bit's status, since it will be reading the stale or incoherent value in the cache rather than the real value being input from the memory-mapped I/O location.

This situation is easy enough to correct by mapping the I/O location into a noncacheable address. The problem gets tricky when an alternate bus master can write into the main memory without CPU intervention. A master is any device which can command the main memory to perform read and write cycles, rather than requiring the CPU to perform these cycles for that device. The most typical examples in single-processor systems are DMA devices like disk controllers or video interfaces.

Taking the disk controller example, visualize, if you will, the CPU initiating a DMA transfer of a portion of a program from a hard disk to main memory at a time after the entire cache has been filled with copies of main memory addresses. Some of these cache locations will doubtlessly be copies of main memory locations which will be overwritten by the incoming DMA data. If these cache locations stay out of sync with the contents of main memory, the newly fetched code will execute in all the uncached addresses, interspersed with cached copies of the old program. As if this were not bad enough, in copy-back caches, outputs from main memory to DMA devices also can cause problems since the data which the CPU thinks it is sending to the DMA output device is the most current, but if a portion of this block is still residing Dirty within the cache, then some stale piece of data will end up going to permanent storage instead. These two problems will be addressed in the following portions of this section.

Perhaps this would be a good place to differentiate between cache policies and cache and/or bus protocols. **Protocol** is the word used by cache architects to express the means by which caches, processors, main memory, and alternate bus masters communicate with each other. The cache policies determine the interaction of the cache with the CPU and the software and set the hit rate of the cache. The protocol is the vehicle by which all the subsystems within the system assure that coherency is maintained and that bus collisions do not occur. In this chapter, either all the coherency mechanisms must be designed around both the cache policies and the bus protocol, or the bus protocol must be designed around the cache policies and the coherency mechanism.

### 4.1.1 DMA Activity and Stale Cache Data

During a DMA transfer, the current bus master (the DMA device) can write into main memory locations which may also be replicated within the cache. The cache controller must be able to assure that the contents of the cache memory continue to faithfully resemble the replicated main memory locations, rather than to contain copies of how the main memory used to look.

The simplest method is called a cache **flush**, and involves invalidating the entire directory every time there is a DMA write cycle. Three of the most common methods of doing this are as follows: 1) to use special invalidate hardware to write an Invalid state to the Valid bit in every cache line, 2) to use a special cache-tag RAM which has hardware reset capability to do the same, and 3) to reset the main Valid flag which gates the tag's comparator output to the CPU (assuming Valid bits are not used in the design). The best that can be said about this method is that it works and is trivial to implement, as long as the cache is a simple write-through implementation. The worst thing about it is that it forces the cache to refill itself after every DMA write cycle. In certain cases this is not so bad. In a PC running DOS, the CPU is always stalled for the duration of the DMA block move, so the added penalty of several subsequent cache line refill cycles is not so noticeable. On the other end of the spectrum, a flush can be devastating in a system using a multitasking operating system like UNIX which attempts to dispatch another task once it requires DMA activity for the first one. The alternate task would ideally be performed out of the cache, so that the DMA activity and the processor activity would not interfere with each other.

Another more efficient way to assure coherency on DMA transfers is to provide hardware that watches all system bus cycles and checks their addresses to alert the cache if one of its locations is being affected.

A **bus watch** or **snoop** mechanism is a simple means of assuring that any pertinent system bus cycle to main memory updates or clears the appropriate cache location. On main memory write cycles, if the addressed location is replicated in the cache, or, in some designs, if that location merely has a high likelihood of being contained within the cache, it will be overwritten or invalidated, depending on the design. A term occasionally used for the process of invalidating a line on a snoop write hit is **back invalidation**, which implies that the invalidation is happening in a direction other than the normal direction in which cache updates occur. Most designers refer to the process as if the cache is watching the system bus and usually use the term **snooping**, while others look at it as if the system bus is looking into the cache and call the process **interrogation** or say that the system bus **inquires** the cache about its contents. Another, less widely used, term is **cross-interrogate**, which sounds like a lawyer's way of describing the process of the current bus

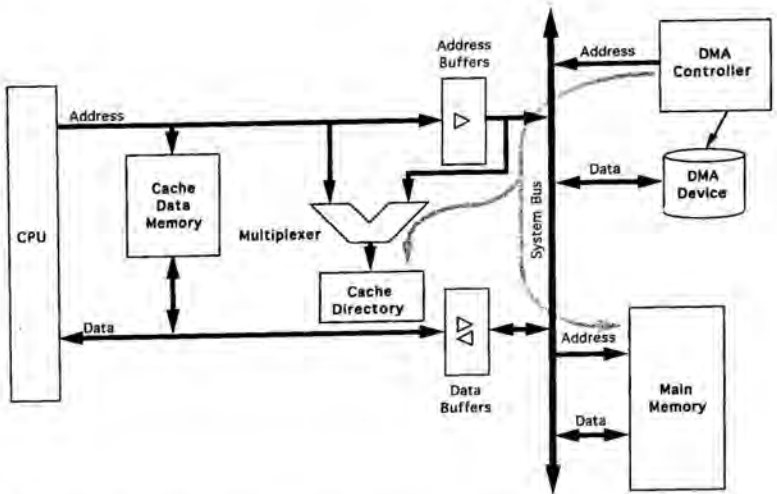
master looking for potential coherency issues within other caches. The problem with the terms *interrogate* and *inquire* is that they imply that the requesting device requests the data first from any existing cache before looking in main memory. In true-to-life designs, caches generally watch the bus and take action when a coherency problem is likely to arise. It's like somebody inside looking out as compared with another person outside looking in.

In a copy-back cache, the snoop logic must also watch out for main memory reads which should be satisfied from a Dirty word in cache, rather than by a stale word in main memory. The DMA device, often a disk, will need to be sent the most up-to-date copy of a memory address, rather than the data which is actually contained within the main memory.

In a snooping cache design, a cache-tag RAM continually monitors main memory bus activity. This can be performed by a separate cache-tag RAM which holds an identical copy or a superset of the cache's actual directory (more on this later) or by the cache's actual directory.

Those designs which use the cache's actual directory fall into two classes. The most common class is one in which the cache is multiplexed between the main memory bus and the processor. This is often called a **dual-ported directory** or a **dual-ported tag**. In some designs, a snoop cycle starts by stopping the CPU, thereby disabling its access to the cache. The DMA address is then routed to the cache-tag RAM, and, in the case of a DMA write to main memory, the location can be either compared and written Invalid (if there is a DMA hit), compared and left Valid, with new data written to the cache data RAM (once again if there is a hit), or simply invalidated (whether or not there is a hit). The choice of these alternatives is another one of the myriad of system-dependent trade-offs, since the first two require read/write cycles, which cause the CPU to be disabled for more precious time, yet the last and fastest alternative will assuredly step on a number of innocent, non-matching cache locations, which will then need to be updated upon a subsequent access cycle, once again costing CPU speed. Just like every other cache decision, there are no easy answers as to which of these three is the best for your system. It all depends on the size and associativity of the cache and, probably to the largest extent, upon the structure of the software being run upon the machine.

Other approaches involve multiplexing the tag RAM in such a way that snoop cycles are invisible to the CPU, or handing the CPU the snoop address and requesting the CPU itself to perform the housekeeping via dedicated hardware, as is done in most processors. The multiplex approach doesn't slow the CPU down, but can only be used if the processor gives the bus up for a significant percentage of the overall operating time. The CPU-based invalidation mechanism can cause significant delays in processing time, as each invalidation usually consumes several CPU cycles.



**Figure 4.1.** Snoop line invalidation using a multiplexed cache-tag RAM. The multiplexer must be inserted into the cache's critical tag address input timing path.

A multiplexed cache-tag RAM in a discrete cache implementation saves little expense and does not reduce chip count because it requires a multiplexer on the address inputs of the cache-tag RAM between the system and local buses (Figure 4.1). The delay caused by this additional logic can prevent a design from keeping up with a fast CPU. This approach does make sense in an integrated cache implementation, like those on most processor chips, since the multiplexer might already be in the critical path and simply need to be widened, and, even if it is not, it could probably be implemented at less than a 1ns speed penalty to the cache-tag RAM's access time.

Another twist to this is referred to as **DMA through cache**, where a DMA device is essentially tied to the CPU/cache bus rather than to the main memory bus (Figure 4.2). During the DMA, the processor is isolated from the CPU/cache bus, and the DMA device reads and writes to and from both the cache and the main memory in exactly the same method as that used by the CPU. Even the cache controller doesn't know that a DMA is taking place. Although this shuts the CPU down for the duration of the DMA, when the DMA is over, the cache is coherent and is likely to contain at least several useful locations, depending on the write miss policy chosen. Probably the best write policy which could be used in a write-through cache of this sort would be to not replace a line on a write miss, since 1) there may have been useful code or data in that line, which the processor would need after



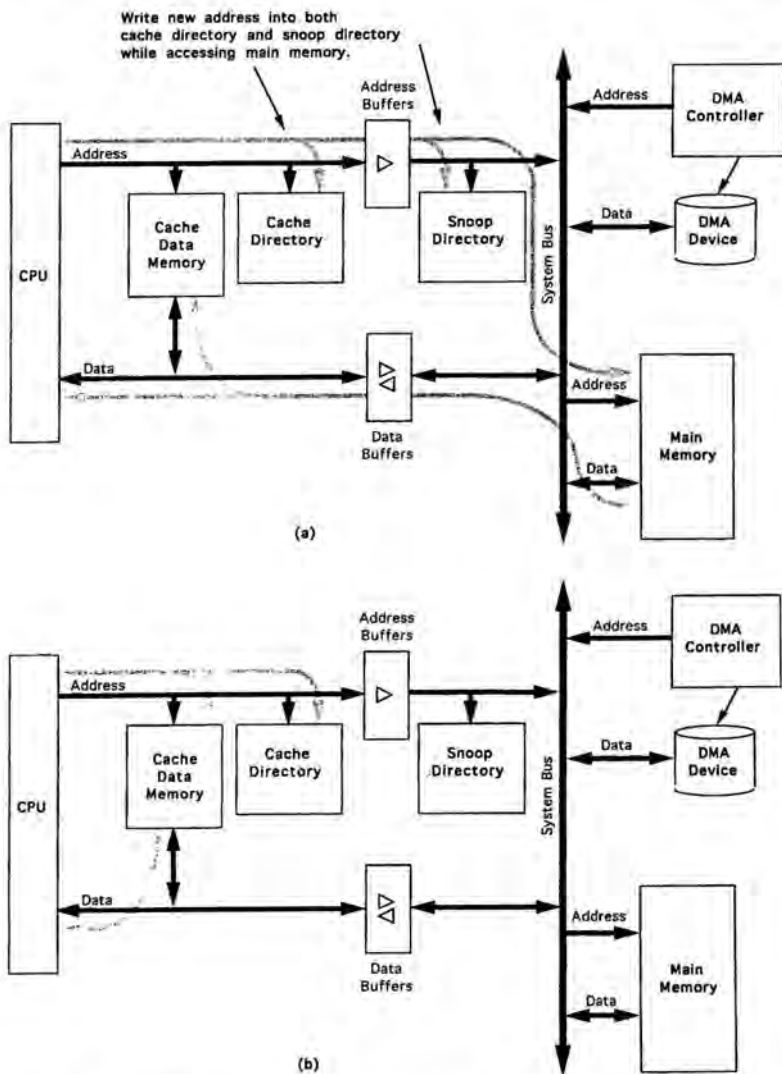
A second class of snoop mechanism uses a duplicate tag to snoop the main memory bus. Depending on the cache design, these can be either very simple, very costly, or completely free but very difficult to understand. Let's move from the simplest to the most complex.

**Dual cache-tag or dual directory** systems offer a higher operating speed, asynchronous operation, and a greater degree of system design flexibility than multiplexed cache systems at a similar chip count, but require slightly more expensive components.

In a dual cache-tag RAM system, there are two identical directories: one which monitors the CPU's address bus and one which monitors the system address bus. A copy of the contents of the local cache-tag RAM is replicated in the system cache-tag RAM. At first guess, it would appear that a special effort needs to be made to assure that the snoop-tag RAM contains the same information as the cache-tag RAM. This is actually a trivial problem. During any cache line update (Figure 4.3), the cache address bus is connected to the main memory bus, so both the snoop-tag RAM and the cache-tag RAM will be seeing the same address at the time that the cache-tag RAM is updated with the tag of the new line. It then becomes simply a matter of using the same write pulse to update the snoop-tag RAM as is used to update the cache-tag RAM.

When the CPU is operating out of the cache and another master write occurs, the system bus address is compared against the address in the system bus cache-tag RAM. If there is a DMA write hit (implying that the main memory address being accessed is copied in the cache), the snoop-tag RAM will notice a match, causing the cache controller to stop the CPU and invalidate or update the matching location. Both the snoop-tag RAM and the cache-tag RAM entries should be invalidated at this time to assure that subsequent snoop hits to a previously invalidated location will not occur. There are even systems which use the same parts, a snoop-tag RAM in addition to a full cache, but upon a snoop hit, flush both the snoop-tag RAM and the directory's cache-tag RAM via their reset input pins. While this is a drastic approach, this system would experience fewer cache flushes than would a system which flushes the directory upon each and every DMA write cycle. It's a little surprising that the flush-on-snoop-hits approach is used at all, given that the complexity of the cache controller is nearly the same for both of these types of coherency mechanisms.

Before we progress to more sophisticated coherency mechanisms, this might be another good place to harp on one of my favorite subjects in this book: the need to consider the software being run on a system when choosing cache strategies. The complexity of the coherency approach really shouldn't be greater than what is required by the operating system being used. Once again, let's use Intel-based, single-processor PCs as an example,



**Figure 4.3.** Using an additional directory to snoop the bus. (a) When a cache line is updated, the same tag and set bits appear on both sides of the data bus buffers, so the snoop and cache directories can be simultaneously updated. (b) When the CPU is not using the bus, simultaneous DMA/snooping and CPU/cache operations can occur.

and let's only consider those which use DMA for disk I/O and nothing else. Should the design be targeted solely at MS-DOS or Windows 3.1 applications, snooping might not be an important feature in a cache design. In simpler operating systems, the CPU is not allowed to operate during DMA accesses, so system performance is not improved by reducing the number of invalidate cycles to the cache during this CPU idle time. On the other hand, in systems using UNIX or other more sophisticated operating systems and in multiple-processor systems, a significant performance improvement can be realized by reducing needless invalidation cycles, since in these cases the processor operates simultaneously with other bus master devices.

The next coherency mechanism in the scheme of ascending complexity involves the principle of **inclusion**. This sort of design is used where the snooping tag is bigger and not necessarily of the same associativity as the cache directory. First, let's see why anyone in their right mind would attempt to do this, then let's explore why it works.

Say you were trying to build an external snooping mechanism for the internal cache of a processor with an 8K-byte, four-Way, set-associative unified internal physical cache, with a line size of four words (16 bytes). It appears that the easiest thing to do would be to replicate the internal cache's directory. How many chips would this take? Putting on our math hats, we see that four separate cache-tag RAMs would be needed, one for each of the four Ways, and that each cache-tag RAM would need to have a depth of  $[8\text{K bytes}/(4 \text{ Ways} \cdot 16 \text{ bytes per line})] = 128$  locations. Assuming that there are 30 address bits, 28 of which are used to address cache lines, and since 7 bits would be used as set bits to get into the 128 tag locations ( $128 = 2^7$ ), the other 21 bits would be the tag bits. Add one Valid bit and the overall cache-tag RAM requirement would be four  $128 \times 22$ -bit snoop-tag RAMs. If we were to use the industry's currently most widely available integrated cache-tag RAM, which comes in an  $8\text{K} \times 8$ -bit organization, a total of 12 devices would be needed to implement the tag alone ( $22 \text{ bits}/8 = 3$ , times four Ways)! By using inclusion, this can be whittled down to two  $8\text{K} \times 8$  integrated cache-tag RAMs hooked up in a simple direct-mapped configuration. Let's see how.

If the snoop-tag RAM can always be caused to contain a superset of the cache's directory, then the snoop-tag RAM can be used to verify all invalidation cycles before these cycles are passed on to the actual cache. This way, a lot of fruitless invalidation attempts to the cache (and subsequently a proportional number of unnecessary CPU wait states) can be screened out and discarded by the snoop-tag RAM. The name "inclusion" indicates that the contents of the entire cache directory are included within the contents of the snoop-tag RAM.

Forcing the snoop-tag RAM to be a consistent superset of a more complex cache's directory may appear difficult at first for a couple of very good rea-

sons. First, a cache with a higher associativity than the snoop-tag RAM will be able to put a copy of a main memory location into any of a number of cache locations, while a less associative snoop-tag RAM will be limited and in certain cases will not be able to concurrently contain a set of addresses which would easily fit within a more highly associative cache directory. Second, the cache may not disclose enough to the outside world to allow the snoop-tag RAM to determine which line is being updated. Most processors do not tell the outside world which internal cache lines are being replaced. If the cache's directory and snoop-tag RAM have such a hard time understanding each other, how can their status as subset and superset be maintained?

The answer is surprisingly simple: inclusion. There are two basic precepts to inclusion:

1. Any tag entry written into the cache directory is also simultaneously written into the snoop-tag RAM.
2. Any tag entry that is removed from the snoop-tag RAM (whether by a bus snoop or by replacement) is simultaneously forced out of the cache.

Astute readers will instantly note that with some processor designs data can be dropped from the cache without the snoop-tag RAM being updated. The snoop-tag RAM will subsequently be prone to experiencing snoop hits on addresses which no longer exist within the cache. This is the reason why the snoop-tag RAM becomes a superset of the cache directory.

Another item which the two rules above don't mention concerns address tags which are put into the snoop-tag RAM, but not into the cache. This might occur if the snoop-tag RAM sported a larger line size than the primary cache. Once again, the snoop-tag RAM in this case simply becomes a superset of the cache directory.

Even if the cache is a four-Way design and the snoop-tag RAM is direct mapped, the four-Way cache can't contain anything with an address not copied in the snoop-tag RAM, since, at the same time that the location was being added to the cache, its address was put into the snoop-tag RAM. And if every address which is invalidated in the snoop-tag RAM is also invalidated in the cache itself (whether or not it still exists within the cache), then there can be nothing remaining in the cache which is suddenly missing in the snoop-tag RAM.

Now that we have assured ourselves that inclusion can indeed guarantee that the contents of the cache directory are a subset of the snoop-tag RAM, we should examine the drawbacks.

Probably the most important drawback is that the cache in a system using inclusion can never stay full. It is inevitable that lines which could have con-

tinued to reside within the cache will be forced out because of problems fitting them within the less-associative snoop-tag RAM. This causes the cache to exhibit a slightly lower hit rate, and will cause a time loss to refetch the invalidated line from main memory if that line is needed again. This is a good reason to assure that the snoop-tag RAM is appreciably larger than the cache directory, so that it reduces the number of lines forced out of itself and subsequently forced out of the cache. On the other hand, if too large a snoop-tag RAM is chosen, more false invalidation cycles will be passed on to the cache, thus slowing the system down. There is no free lunch! (As an aside, we have not considered how small the snoop-tag RAM can get. Believe it or not, the snoop-tag RAM could be *smaller* than the cache-tag RAM and inclusion would still work. If the only items which are allowed to reside in the cache have addresses which are copied within the snoop-tag RAM, then only a fraction of the cache-tag RAM *smaller* than the snoop-tag RAM will end up being used in such a system. This would work, but would be a shameful waste of the unused portion of the cache.)

A second apparent drawback is that inclusion may dramatically increase the number of invalidation cycles presented to the cache. Each and every time an address is removed from the snoop-tag RAM, a cache invalidation cycle occurs. This will be the case whether or not the address being removed from the snoop-tag RAM actually still exists within the cache, and whether the snoop-tag RAM entry was invalidated due to a bus snoop cycle, or because of a cache line update which caused a snoop line to be overwritten. Although this looks bad, it is really a strong advantage of inclusion.

Most cache invalidation cycles will be triggered by the replacement of a valid entry in the snoop-tag RAM. This only occurs during a cache miss, and, during cache miss cycles, the CPU is stopped and cannot proceed until it receives the new data. Since an entry is being replaced in the snoop-tag RAM, several cycles will pass before the new data is available to the CPU, and this wasted time can be used productively to invalidate a line within the cache itself! In this way, most of the cache invalidation cycles are forced to coincide with a time when the CPU would ordinarily be stopped. Only those bus snoop cycles which pass through the snoop-tag RAM screen are permitted to stop the CPU in order to cause a performance-threatening invalidation cycle.

If a secondary cache is used in a system, inclusion can be used to allow the secondary cache to prescreen bus traffic and limit the number of false invalidation cycles which are passed on to the primary cache. The secondary cache's tag can get a second use as a snooping tag at a very small cost. A negligible quantity of additional logic is usually required to support inclusion using an existing secondary cache. The set bits output from the primary cache/CPU subsystem will contain the same value which is to be driven into the primary cache to invalidate the primary cache line, so a set address latch is needed. This latch can be absorbed into the lower bits of an existing address

buffer, so the cost in additional chip count is usually zero. The tag bits required by inclusion are supplied from the secondary cache's directory. Very few additional terms usually need to be added to the cache controller state machine, so the cache controller's complexity is not dramatically increased.

**Dirty inclusion** is similar to inclusion for cache hierarchies in which the upstream and downstream caches both use copy-back write strategies. With Dirty inclusion, a secondary copy-back cache would contain a line marked Dirty for every line marked Dirty in the primary cache; however, some Dirty secondary lines would not necessarily be marked as Dirty in the primary cache. This allows a line to be evicted from the primary cache without causing any write traffic to occur on the interface between the secondary cache and the main memory. The secondary line does not get evicted until the secondary cache needs the line to accept new data, and secondary line updates are less frequent than primary line updates, so the traffic stays down. Another advantage of using Dirty inclusion is that snoop read hits to a Dirty line in a copy-back cache must be supported from the cache rather than from main memory. If the secondary cache is to prescreen bus write cycles before allowing invalidate cycles to pass through to the primary cache, then it also makes sense to allow the secondary cache to prescreen read cycles. This can happen if the status of every secondary line is the same as the status of the primary line. This is not to be construed to mean that the Dirty line in the secondary cache would always contain the same data as the Dirty line in the primary cache. For this to happen, the primary cache would have to be write-through and would have to update the secondary cache every time the CPU updated the Dirty line. Instead, the status of the secondary line is updated to Dirty when the primary line is first taken Dirty, and maintains the status for snoop cycles, even though the data from the secondary cache cannot be guaranteed to be coherent in the event of a snoop read hit. Therefore, all snoop read hits to Dirty secondary cache lines must be sent to the primary cache. Still, this beats sending all bus read cycles to the primary cache.

#### 4.1.2 Problems Unique to Logical Caches

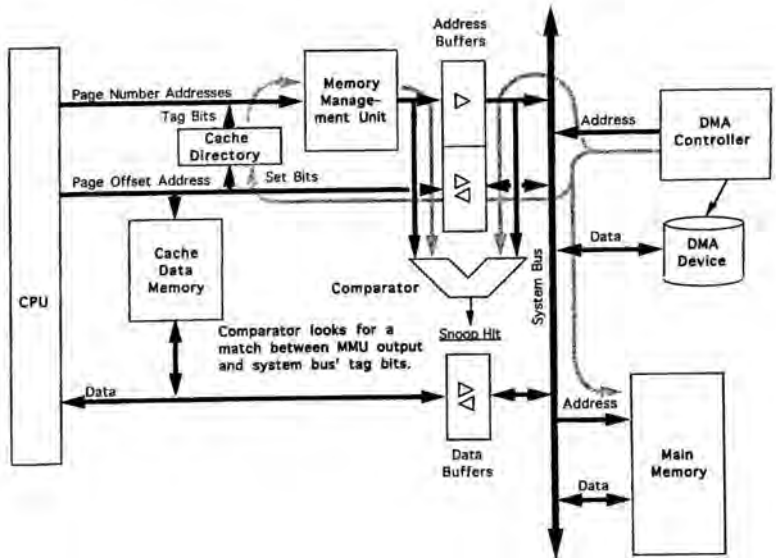
Probably the biggest difficulty encountered in logical caches is the problem of address aliases, first described in Section 2.2.1. The difficulty stems from the fact that two separate virtual addresses may be mapped into the same physical address. Another way to say this is that two virtual addresses which share the same offset addresses but have different page addresses might be mapped by the operating system into the same physical page and would therefore share the same offset within the same physical page (in other words, the same physical address). This is one way that tasks in a multitasking system communicate with each other (for example, system calls from an application program).

During any DMA write into main memory, cached copies of the address written into by the DMA device must also be either updated or invalidated. If two virtual addresses are mapped to the same physical address, there is the possibility that there will be two copies of the same main memory location in the cache. Both will become stale when another bus master writes to the main memory.

Of course, there are lots of ways to solve this problem, the least sophisticated of which is to flush the cache upon every DMA write cycle. More elegant solutions involve disallowing the cache from using as set address bits any CPU output addresses which lie within the page number bits of the MMU. Although this puts a severe limitation on the size of the cache in many cases, it completely does away with the possibility that within a single Way or cache (in a nonunified cache design) there will be two concurrently cached copies of the same physical address which represent two different logical addresses. Think about it. The only way that two separate logical addresses could be mapped to the same physical address is if the two logical addresses had the same offset bits but different virtual page numbers. A large cache, which used some page number bits as set bits, might be able to maintain two different copies, at two different set addresses, of the same physical address, but a cache which had its set bits limited to be a subset of the offset addresses could never have two copies of the same physical address since they would both end up being mapped to the same set address.

The other rule in applying this approach is that only a unified, direct-mapped cache should be used in a snooping logical cache design. During a DMA, the set bits of the DMA address can then be fed into the cache-tag RAM's snoop mechanism, and the tag outputs that set address's virtual page number (see Figure 4.4). This can only work if the set bits are the same on both sides of the MMU, agreeing with the restriction requiring the set bits to be a subset of the page offset address bits. The MMU is then called upon to translate the tag's virtual page number output into a physical page number which can then be compared with the DMA's tag address bits. If a hit is discovered, that cache line is invalidated. If this sounds like a slow process, that's because it is, but think of how much slower it would be if two snooped tags had to process their tag bits one Way at a time through the MMU in a nonunified cache or a multiway design! One alternative is to invalidate all entries with matching set address bits, but this might be difficult to implement and would needlessly invalidate a disproportionate number of innocent bystanders. It is nearly impossible to perform a clean snoop invalidate process in any way which does not require the CPU to stall during the entire snoop cycle, so logical snooping cache designs are usually kept simple just to avoid large speed penalties during snoop cycles.

Although I have never seen such a machine, I would not doubt that some



**Figure 4.4.** Snooping addresses in a logical cache. The set bits of the DMA address are put onto the CPU bus, and the cache-tag RAM outputs that set's logical tag bits. This logical tag is then translated into a physical page number by the MMU, and the page number is compared with the corresponding bits of the DMA access (a snoop hit).

are implemented with reverse translation schemes to map physical addresses back into all the relevant virtual addresses, then to allow several Ways of a highly associative cache to snoop the newly created virtual address. This would allow the logical cache to be nonunified or more associative and would also remove the cache size restriction caused by requiring the set bits to be a subset of the page offset bits. I just hope nobody I know is saddled with the job of designing such a monster.

Discussion of the aliasing problem often centers around context switches, which are more of a software way of describing those times during which DMA activity is likely to occur. Most concern about aliases in general is centered around context switches in both cached and uncached systems, since the MMU mapping of main memory and disk is very similar to the mapping used between the cache and the main memory. If you have the opportunity to discuss your cache design with the software designers who conceived your operating system, go ahead! You'll both be surprised at how many problems you have in common, and you'll probably help broaden each other's perspective.

## 4.2 MULTIPLE-PROCESSOR SYSTEMS

In tightly coupled multiple-processor systems, especially those in which any processor can perform any task in any part of memory, the problems of cache coherency are pretty big. Just as a refresher, we'll go over the nomenclature *tightly coupled* and *loosely coupled* again, since they were first defined in Chapter 1. A tightly coupled multiprocessing system is designed to allow main memory to be equally accessible from each processor. Loosely coupled systems, on the other hand, are made of two or more processors, each of which has a private memory space. Examples of both sorts of system are shown in Figure 4.5. The connection between the processors in a loosely coupled system can be a shared portion of main memory, FIFOs, dual-ported memory, or even a serial link like a local area network, or a dedicated interprocessor communication channel, as is used on the Inmos Transputer. Hypercubes use loosely coupled architectures. Since the processors in a loosely coupled system each have their own main memories, there is absolutely no reason why any memory locations in the two individual main

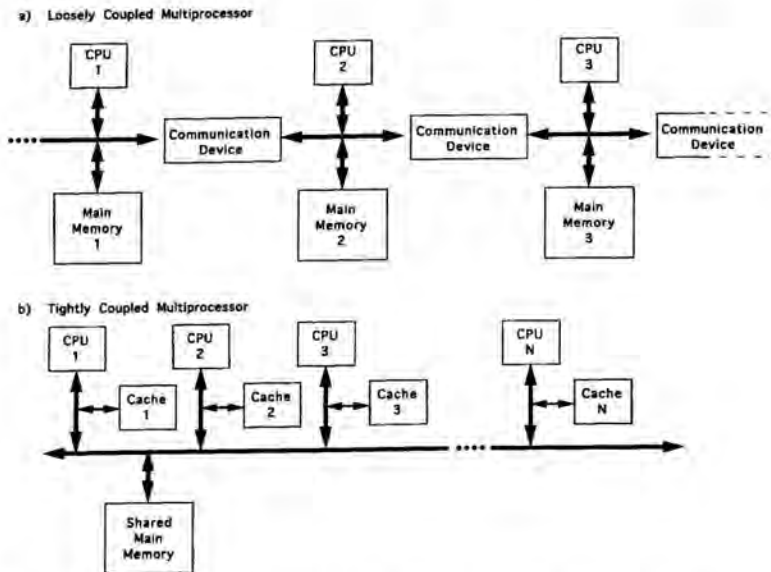


Figure 4.5. Examples of loosely coupled (a) and tightly coupled (b) multiprocessing systems. Main memory is shared in tightly coupled systems.

memories should need to match each other, so no steps need to be maintained to assure cache coherency beyond those which would be required in a single-processor system to avoid incoherent cache copies of the input/output activity of the processor.

One very simple means of assuring that caches in a tightly coupled system stay coherent is to set up a noncacheable space in memory called the **coherency domain**. Since the data in the coherency domain is not cached, that data is always the most current. This is a communication space, which is arbitrated between processors. The coherency domain approach was used by systems like the Honeywell Series 66, early versions of the Intel i860, and the Elxsi 6400. This approach is also available via a status bit in the MMU's page descriptors in certain microprocessors. The major drawback of using such a scheme is that it becomes restrictive to the programmer, since all programs must waltz around the coherency domain with any piece of data or code which is not intended to be shared. Invariably, the portion of memory set aside for the coherency domain will always be found to be too small for certain applications and will consume too large a portion of the memory space for others. Naturally, coherency domain approaches can only be used in systems where the software is defined at the same time as the cache architecture, rather than the cache being designed to accelerate the performance of an existing CPU/software combination. This occurs all too rarely.

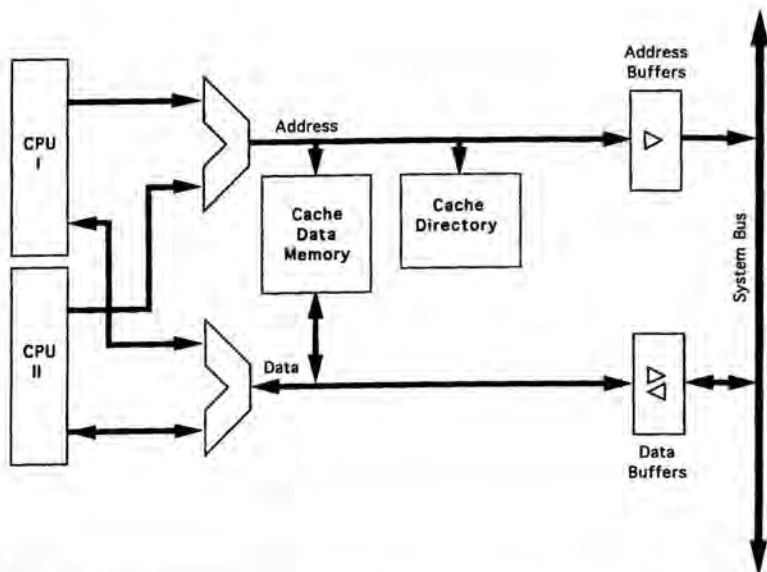
There are other, more flexible approaches available if the hardware and software are being defined at the same time. One of these is the use of a **sophisticated** protocol, in which a customized compiler generates and stores bits corresponding to different main memory addresses to tell the cache which main memory spaces are cacheable and which are shared. The use of the special compiler disallows these caches from being software transparent, so they are not selected for use in systems where source for the code to be used is unavailable. Those protocols which do not depend on compiler support are called **naive** protocols and will make up all of the examples given at the end of this chapter. Naive protocols are the more useful of the two in the widest variety of applications, but they are also harder to understand than sophisticated protocols.

#### 4.2.1 Two Caches with Different Data

It's pretty obvious that a tightly coupled system with two or more cached CPUs can run into situations where the two CPUs' caches each have a copy of the same main memory location. One of the main reasons to use a tightly coupled architecture rather than a loosely coupled architecture is to allow arbitrary memory locations to be shared between the different processors, and these shared locations will perform more slowly if they are not allowed to be cached.

In Section 4.1.1, we looked at the problem of assuring that DMA write cycles update their appropriate cache locations and explored several of the means used to implement coherency. In tightly coupled multiprocessor systems, similar means must be taken to assure that any main memory write cycle from one processor updates the other processors' caches where appropriate.

One method which has somewhat fallen from grace is the use of a single cache to support multiple CPUs, as is shown in Figure 4.6. The Univac 1100/82 used such an approach to support two processors, and Futurebus+ allows for such a configuration. This can be made to work if the CPUs can be interleaved with each other; however, today's and tomorrow's CPUs often run at such high speeds that no discrete primary cache could be designed to be fast enough to service the bandwidth needs of the interleaved CPUs. Another problem is that a dual processor using the same cache is about twice as likely to thrash as a single processor operating out of the same cache. Last, in systems using on-chip primary caches, a shared secondary cache is sometimes used to reduce bus traffic. If all of the CPUs in the system are tied to the same cache, the bandwidth limitation simply gets moved from the CPU-main memory interface to the cache-CPU interface, rather than being reduced.



**Figure 4.6.** One cache serving two CPUs. This tends to cause thrashing since the two CPUs are in contention over the allocation of the cache.

### 4.2.2 Maintaining Coherency in Multiple Logical Caches

Coherent logical caches in single-processor systems have a lot of restrictions, which were outlined in Section 4.1.2. The same restrictions apply to logical caches in multiple-processor systems. The most important is that the cache set address bits must consist of a subset of the MMU's page offset bits of the memory-mapping scheme, unless the designer wants to go to some pretty extreme measures to maintain coherency.

Let's look back at the mips R4000 as an example of how extreme those measures can become. Read this closely, because the scheme used in the R4000 is very involved. The R4000 was designed for multiprocessor systems and used a logical primary cache and a physical secondary cache. The secondary cache screened primary snoop invalidates through inclusion. The primary cache was eight times larger than the size of a page of main memory, so three of the page numbers in the MMU were mapped into the set bits. This implies that there can be aliases within the primary cache for which the secondary cache must snoop.

A drastic way of solving this problem would have been to invalidate all primary cache locations which had any likelihood of matching an updated main memory location. On a secondary snoop hit, all right of the primary cache locations with matching page offset address bits would be invalidated. This would have worked, but would reduce the performance of the cache by causing unnecessary invalidations.

The way mips solved the problem was to make a physical secondary cache a necessary part of the system and to use inclusion, assuring that every primary cache location was replicated within the secondary cache. Each of these secondary cache locations contains three **color** bits, directing the secondary cache to the corresponding primary cache location. These three bits are stored at the secondary cache line's set address, just like the tag bits and the line status bits.

When there is a secondary snoop hit, the secondary cache merges its three color bits with the page offset bits to create a logical set address of the matching primary cache address. The primary cache location at this logical set address is then invalidated. Multiple cached copies of the same main memory locations cannot exist in a system using this sort of inclusion, since both logical copies would map to the same physical secondary cache address, and there is a one-to-one mapping of primary cache locations to their secondary counterparts. If there were not this one-to-one mapping, each secondary line would have to store multiple coexistent sets of color bits to account for every possible alias.

Let's see what would happen if the CPU attempted to put aliases of a location into its primary cache. Say there exist primary and secondary cache copies of a main memory location, and the processor wants to read (and

cache) a different logical address which will map into the same physical address. This will be treated by the primary cache as a miss/refill cycle, and by the secondary cache as a hit. The color bits are checked against the three lower logical page bits and will be found not to match, so the secondary cache would treat the cycle somewhat like a miss cycle, performing its inclusion housekeeping by invalidating the primary cache location which has matching offset and color bits, yet the secondary cache's copy would not be invalidated or evicted. A new copy of the secondary cache data is then copied into the new primary cache address, and the only part of the secondary cache line which would be modified are the color bits, to reflect the new logical address which that cache line represents. No main memory accesses take place, even if the secondary cache's line contains modified data. As I said before, this is complex and might not be deemed worth the effort.

### 4.2.3 Write-through Caches as a Solution

The problem of assuring that two or more caches in a system stay coherent with each other really boils down to the issue of how to handle write cycles. DMA writes into main memory must be accommodated by the individual caches, and writes performed by another processor must not disagree with the data subsequently held in another cache for the same memory location.

A very direct means of assuring that this happens is to cause all write cycles, whether from a DMA device or a cached processor, to be placed on the main memory bus. Write-through caches send all of their write cycles to main memory, either directly or through write buffers, so they are an obvious choice for this sort of system. This approach is the one used in IBM's 3033 processors, as well as Sequent's Balance 8000 system. All write cycles on the bus are snooped by all caches, and write snoop hits will cause any cache with a matching address to invalidate or replace its copy of the data being updated. As previously mentioned, some systems flush the entire cache on a write snoop hit.

This last option is even less desirable in a multiprocessor than in a DMA system, since write cycles from other processors will be small, frequent, and random in nature, rather than the occasional large block data transfer which is typical of DMAs. If each bus write caused all caches in the system to flush, the number of cycles which would be satisfied by any processor's cache would be almost zero.

One problem revolves around the use of write buffers. Assume that a processor writes data for a certain address into a write buffer, and another processor is simultaneously reading from the same address in main memory. Should the second processor first receive the now-stale data from main memory, with the data being invalidated or updated immediately after its

use, when the write buffer is finally granted access to the bus? In most designs, this is not really a problem, since the processors are not synchronized to begin with. Other designs, however, shorten the response from one processor's write to another's read by configuring write buffers to snoop the bus, just as a cache would, and to either abort the other processor's read cycle until the write buffer is emptied into main memory or to supply the data directly from the write buffer to the requesting processor. A third option is to grant write buffers higher priority during bus arbitration than is granted to any reading device. This will always allow the write buffers to empty before any possible read conflict arises.

#### 4.2.4 Bus Traffic Problems

Now that we've seen the advantages afforded by write-through caches from the perspective of multiprocessor coherency problems, let's look at the same caches from the perspective of bus traffic. Bus traffic is a real concern and is often a reason to add a cache to any system, especially multiprocessor systems. Naturally, the less bus traffic, the better, but by no means does the designer want the system's bus to come anywhere near saturation.

First, let's work under the assumption that all bus cycles occur with no wait states. This is just a temporary assumption to clarify. We'll return to reality a little later. As we have done before, let's assume that 10% of all CPU cycles are write cycles, and since the caches used are write-through, all of these writes are propagated to the main memory bus, both to be written to the main memory and to be snooped by all of the other caches in the system. Let's also assume a 5% read miss rate on the cache, and since the read cycles make up 90% of the CPU's I/O cycles, read misses will account for 4.5% of all CPU cycles.

Write cycles and read misses will occur pretty randomly. If they were completely predictable, the system would quickly synchronize itself, and up to six processors ( $100\% / [10\% + 4.5\%]$ ) would run as fast as they could on such a bus before there were any signs of a problem. This does not happen in the real world, though, and every added processor will cause arbitration and associated overhead, so the bus will cause added processors to show diminishing returns long before six processors are installed. (Still, remember that we are basing this analysis on some assumed numbers. These numbers are extremely dependent upon the cache design, as well as on the software being executed. Sequent's Balance 8000 was said to be able to sustain up to 12 CPUs before saturation, and it used write-through caches.)

Now, let's see what happens when a wait state is added. Assuming this means that every bus cycle is twice as long as it was in the previous case, the bus would quite simply become saturated with half as many processors. Di-

minishing returns would most probably become glaring even when the second processor was added.

Just to add insult to injury, let's now consider the fact that all of these main memory bus cycles will be in contention with each other for bus usage, further piling up the wait states. Most multiprocessor bus arbitration algorithms cost at least one extra wait state, even if the bus doesn't happen to be in use at the time.

I'm sure that you are looking at the example and saying "There's a lot to be gained by increasing the read hit rate and by making sure that the main memory system is as fast as possible," and you're right, but high-speed systems are nearly impossible to design with zero-wait buses, and read hit rates can only be pushed so far. What's the next option? Well, since the write cycles are the most frequent users of the bus (10% vs. 4.5% in this example), doesn't it make most sense to go after them?

Say your write-through cache, with its 5% miss rate, can be converted to a copy-back cache with a similar write miss rate. Suddenly, the bus is needed not for all writes plus 5% of the reads for 14.5% of all CPU I/O cycles, but for only 5% of all CPU I/O cycles. This looks well worth the effort, but in a multiple-processor design it can cause a lot of difficult problems. The next section will address these.

#### 4.2.5 Copy-back Caches and Problems Unique to Copy-back Caches

We have now decided that bus traffic would have a far lower impact in a system if it could be reduced by using a copy-back cache, so we want to know if there are any special considerations which would cause difficulties in a copy-back multiprocessor cache design. There sure are!

By their very nature, copy-back caches maintain data which is allowed to be incoherent with the associated main memory location. How in the world can a cache location be coherent from cache to cache, yet be incoherent with main memory? Well, maybe it doesn't always have to be! There is every possibility that a written location belongs solely to a single processor, for example, a loop counter or a private pointer. The locations which are shared between processors, however, must always match from cache to cache. Although this seems like a place where the code would need to be written to support the protocol, this is not the case, and we will soon see that the method of assuring that the shared write cycles are appropriately communicated, while private writes are not, is not as difficult as it may seem.

Just to take a quick break from all of this complex talk about multiple processors, let's look at a simple software-controlled mechanism for assuring that DMA read cycles use the most up-to-date copy of a piece of data in a uniprocessor system with a copy-back cache. This problem is trivial in com-

parison with the problems encountered in a multiple-processor system, since the CPU initiates all DMA activity, and the process of assuring coherency can be tacked onto the routine which allows the DMA activity to start. In multiple-processor systems, the communication between processors is less tightly controlled and can happen more or less randomly in bits and pieces.

To support DMA coherency in uniprocessor systems, some processor instruction sets provide the user with an instruction which initiates a copy-back cycle on all Dirty cache locations. This is typically called a cache **purge** cycle, since the process purges all of the Dirty locations out of the cache. If the program controlling the start of a DMA from memory (possibly a write to disk) commences by performing a purge instruction, there will be no Dirty locations when the next instruction is allowed to execute. Intel's IntelArchitecture, with its internal write-through cache, provides a write-back invalidate data cache (WBINVD) instruction to support external copy-back caches, but all the instruction does is to raise a flag and expect the external cache controller to stop the processor from executing any further I/O activity until all Dirty external copy-back cache lines have been updated in main memory. Cache designers are sometimes frustrated at the duration of an instruction which must inspect the status of every cache line and might even need to perform four or eight times as many write cycles as there are lines in the external cache! Worst yet, some processors support an **export** instruction, which is aimed at turning the purge process into a software loop, rather than a hardware purging mechanism. The export instruction, when used in a purge loop, will send a copy of a single Dirty line within the cache back to main memory, so that that individual main memory location is made coherent in main memory. Naturally, purging the cache under software control does nothing to solve the random snoop read hits by alternate processors in a multiple-processor system, so let's move onward to the more challenging task of assuring coherency via hardware for arbitrary coherency conflicts.

In the write-through design suggested in Section 4.2.3, main memory was viewed as the holder of "the truth." Any cache looking for data could always go to the main memory to get the most current copy of the data. In a single-processor copy-back cache design, the most current copy of a main memory address might be held in either the main memory or in the cache, and the location of the most current version is indicated by the cache line's Dirty bit. Obviously, the location of the most current version becomes more obscure in a multiple-processor system.

Something which might instantly spring into your mind is that, if replicas of all the tags and Dirty bits for all caches were available in a centralized location, then each cache which was performing a read cycle would be able to direct its read cycle either to the main memory or to the cache which held the most current copy of the data. CDC computers and Chips and Technologies' M/PAX multiprocessing chip set operate this way, in what is re-

ferred to by the following variety of names: **directory-based**, **memory-based**, **global directory**, or **memory tagging**. In a directory-based system (Figure 4.7), every memory location or group of memory locations which has the same length as a cache line (sometimes called a block, and sometimes not) will also contain one or two extra bits per processor. These bits indicate the status of that memory block as a cache line in one or more caches. The status might indicate that processors A, B, and F contain a copy of the data and that it has not been modified. In another block, the status bits might indicate that only processor D has a cached copy, and this copy is the only valid one in the system. In some more elaborate systems, the status bits might indicate that D, E, and F contain valid copies, but the copy in main memory is not coherent with these. All of these status bits must be reset upon system initialization.

In any case, the main memory acts as a referee and must be constantly aware of the status of the caches of each processor. This simplifies the design somewhat, since all monitoring functions are assumed by a single unit. On the other hand, the number of processors used is not arbitrary, but is limited by the number of main memory bits allotted to keeping the status. It might be tempting to put these bits onto the processor board, so that processors could be added indefinitely at will, but then expansions and reductions in main memory size would require simultaneous modification of all processor boards.

This approach is not quite as simple as the preceding text might suggest, because some means must be chosen to disallow the concurrent existence of two or more copies of the same main memory location during a time when one processor wants to write into its cached copy of that location. If

| Processor Bits |       |       |       |       |       | Data Bits  |   |        |                              |   |  |
|----------------|-------|-------|-------|-------|-------|--|---|--------|------------------------------|---|--|
| A              | B     | C     | D     | E     | F     | Each main memory location is equivalent in size to a cache line. |   |        |                              |   |  |
| Valid          | Dirty | Valid | Dirty | Valid | Dirty |  |   |        |                              |   |  |
| 1              | 0     | 1     | 0     | 0     | 0     | 0  | 0 | Line 0 | Valid copies in A, B, and F. |   |  |
| 0              | 0     | 0     | 0     | 0     | 1     | 1  | 0 | 0      | 0                            | D has only valid copy, and has written to it. |  |
| 0              | 0     | 0     | 0     | 0     | 0     | 1  | 1 | 0      | 1                            | 0   | D, E, and F have copies, and D has written to all. |
| 0              | 0     | 0     | 0     | 0     | 0     | 0  | 0 | 0      | 0                            | 0   | No cached copies exist.                            |
| 0              | 0     | 1     | 0     | 0     | 0     | 0  | 0 | 0      | 0                            | 0   | Cache B has only copy.                             |
| 1              | 0     | 0     | 0     | 0     | 0     | 0  | 0 | 0      | 1                            | 0   | Valid copies in A and F only.                      |
| 0              | 0     | 1     | 0     | 1     | 0     | 1  | 0 | 1      | 0                            | 0   | All except A have valid copies.                    |
| 1              | 0     | 1     | 1     | 1     | 0     | 1  | 0 | 1      | 0                            | 0   | All have copies, and B has written to all.         |
| •              | •     | •     | •     | •     | •     | •  | • | •      | •                            | •   | •  |
| •              | •     | •     | •     | •     | •     | •  | • | •      | •                            | •   | •  |
| •              | •     | •     | •     | •     | •     | •  | • | •      | •                            | •   | •  |
| •              | •     | •     | •     | •     | •     | •  | • | •      | •                            | •   | •  |
| •              | •     | •     | •     | •     | •     | •  | • | •      | •                            | •   | •  |

**Figure 4.7.** Organization of main memory in a memory-based coherency architecture. Bits within main memory keep track of the location and coherency of cached copies of each line-sized main memory location.

all cache writes first had to check for copies of the same line in other caches, the main memory bus would be saturated with cycles interrogating the status of various lines of main memory. For this reason, many protocols disallow the existence of multiple cached copies of a Dirty line.

A much more popular method of assuring coherency in multiple copy-back caches is to use the inherent snooping mechanisms built into each cache. These systems are called **cache-based**. Several recipes can be used to assure that data will remain accounted for and that there can never be more than one copy which the entire system agrees is the most current at any particular instant. Any cache line's state can be changed by either a CPU cycle or by a snoop cycle.

The most widely used multiprocessor copy-back cache coherency protocols are those dubbed **write-once**. Some call these protocols **write-first**. In write-once protocols, the first write cycle to a location held in cache is also written onto the main memory bus. In other words, a cache line is treated as it would be in a write-through cache for the first write cycle. Why is this done? Well, it allows the other processors to snoop the write cycle and invalidate their own copies of the same line. It is key to this scheme that they invalidate their copies rather than update them, since this will be the only bus write the writing processor allows to go onto the bus. A processor's bus traffic will be a little higher for a write-once cache than it would be in a standard copy-back cache, but this is one of the penalties of maintaining coherency.

A processor must perform two kinds of writes. The first is the write to a private location, which, as we saw, would be something other processors would not want to see, like a loop counter. The second kind of a write would be for public data or data other processors would want to see. In a processor using a write-once protocol, the private data write follows a sequence where the processor writes to the bus the first time, invalidating any other cached copies of the same memory location, then, during all subsequent write cycles, writes only to the cache, at the cache speed rather than the main memory speed. Bus traffic would also be eliminated for all of these subsequent write cycles. The Dirty line is copied back into main memory only if it gets evicted during a line refill. Another way to look at this is that the cache controller assumes that the data is public on the first write cycle, but if it is able to write to the same location twice without intervening requests for the same information from other devices, then the controller determines that the location is private.

A write cycle to a public data location would unknowingly proceed in the same manner until another processor requested the data. If only the first write cycle had taken place, the other processor would update itself from main memory, and the processor which had performed the first write would be told that the first write was no longer the first write. Another write-once cycle would need to be performed to invalidate the second cache's new copy of the matching main memory location's data. If more than one write cycle

had taken place before another processor requested a copy, then main memory would not be current, and the requesting cache would need to procure a copy of the contents of the cache line into which the first processor had been writing. The means by which this happens is called **intervention**, since the cache with the most current data must intervene, forcing the reading device not to see the current contents of main memory, but to see the updated data. Means of intervention will be investigated shortly.

It would appear that yet another status bit would be required on each cache line to tell the cache controller whether the write cycle was the first or some subsequent write cycle to this cache line. This is not the case, since a typical copy-back cache has all of the mechanisms to support an unused state. Recall from Section 2.2.4 that a typical copy-back cache uses two status bits: Valid and Dirty, yet there are typically only three states indicated by these bits: Invalid (clean or dirty), Valid Clean, and Valid Dirty (Figure 4.8). What if we had a more elaborate interpretation of the two bits which allowed us to use the spare Invalid state to keep better track of the cache line's status? This would allow us to store four states: Invalid, Valid-and-never-written-to-by-any-CPU, Valid-and-written-to-once-by-this-CPU, and Valid-and-written-to-more-than-once-by-this-CPU. (Cache designers don't use this terminology, but have far more brief nomenclature which we will examine shortly, in Section 4.3.) By doing this, we can no longer refer to the bits as Valid and Dirty, since their meanings are now encoded, but we still have the same number of bits, and the changes to the cache controller to support this new protocol are simple. The only state which requires the cache controller to intervene is the state in which the cache's copy of a piece of data is more current than that held in main memory. This is the Written-to-more-than-once state, which is sometimes referred to as the **Private** state, since it is the only state in which the data in the cache is not shared or coherent with the data in either main memory or another cache.

A protocol which offers a slight twist on the write-once algorithm is called **broadcasting** or **write broadcasting**. As opposed to the invalidation which is caused to happen by the first write in a write-once scheme, broadcasting cache protocols allow the recipients of the newly written data to keep copies in their caches of the data which was written to the bus by other CPU/cache combinations. Because of this, broadcasting caches must be supported with system bus cycles which differentiate between a broadcast write and a write which is intended to invalidate other cache's copies of the same line. Furthermore, a whole new set of states is required to account for the new approach. Some broadcasting protocols will be examined in later sections.

There are two kinds of intervention: **indirect data intervention** and **direct data intervention**. Both deserve thorough explanations. We'll start with indirect data intervention, which is by far the simpler of the two.

| Valid Bit | Dirty Bit | Status   |
|-----------|-----------|--|
| 0         | X         | Invalid Line                                     |
| 1         | 0         | Valid Clean Line. Matches main memory.           |
| 1         | 1         | Valid Dirty Line. More current than main memory. |

Figure 4.8. The three states in a typical copy-back cache design.

As previously stated, the problem to be solved exists when a cache detects a read snoop hit to a location which is marked Dirty. Indirect intervention caches will abort the snooping read cycle, gain control of the main memory, write from the snooped address to the appropriate main memory location (without CPU involvement), updating the line's status from a Dirty state to a clean Valid state, then relinquish the bus so that the other CPU can again attempt, and succeed, to read the data it needs. (The process of updating main memory during a snoop cycle is sometimes called **XI castout** for "cross-interrogate castout." Cross-interrogation, as we have already seen, is another term for snooping, and castout is actually eviction, but the creators of the term XI castout probably don't really mean eviction; they probably perform main memory updates and modification of the line's status to a Valid-and-never-written-to-by-any-CPU state, just like you and me.)

The reason this intervention is indirect is that the request for data in the snooped processor's cache is answered not by the snooped cache handing the data directly to the requesting device, but by the process of handing the data through main memory, a more indirect path. This is reminiscent of conversations in some old movies, where three people are in a room, and one tells another to tell the third something because the two of them are not on speaking terms. Some indirect data intervention protocols allow a device which has had its request aborted to watch the bus traffic and, if a write cycle to a matching address occurs, to grab a copy and continue with its read cycle, saving time and reducing bus traffic over protocols which require the read cycle to be retried.

Indirect data intervention does require special bus support. Either of two methods are used to abort bus cycles, depending upon whether the bus protocol does or does not support split transactions, and a mechanism must be provided to allow main memory updates from snooped locations to take priority over any other cycle from another processor, even if the other processor would normally be granted a higher priority in bus privileges.

In split-transaction systems (first defined in Section 2.1.2), a bus master

issues a request and removes itself from the bus, awaiting a response from the executor of the task. The snooped cache in split-transaction systems sends an abort/retry message to the requesting device, then gains control of the bus before another request can be issued.

In systems without split transactions (where a bus master issues a request and stays on the bus until a response is recognized), either a higher priority request forces the requesting processor off the bus and puts it into a Hold state until the main memory update is complete, or an abort/retry signal causes the cycle to be restarted, but in the intervening time, a higher priority master, the snooped processor, gains control of the bus to update the main memory. It's pretty obvious that if the read cycle comes from a processor with a higher priority in attaining the bus than the snooped cache, and if the response to the snoop cycle is not allowed to temporarily take a higher priority, then the system will lock up while the higher priority processor waits for the data which the lower priority processor is not allowed to place into main memory.

Direct data intervention systems allow any responding cache to disable system memory on a bus read snoop cycle, and allow the data from the snooped location to be placed on the bus supporting instant communication between the snooped cache and the requesting processor. This is also called a **cache-to-cache transfer**, since main memory is not involved in the transaction. In these systems, either of two methods can be chosen for the problem of multiple cache copies of data for which there is no valid copy in the main memory. The simplest to understand is the use of **reflection**, a process by which the main memory has a snoop mechanism similar to the snoop mechanism on the caches of the other CPUs on the system. The snoop mechanisms on the other processors would typically use cache-tag RAMs to monitor the bus for matching address cycles. The main memory, on the other hand, consists of a single large, contiguous block of addresses, so all that it requires is an address decoder, in fact the same address decoder which is used to assign the memory space that the main memory would ordinarily be mapped to if not superceded by an intervening cache. If a processor starts a read cycle to an address which is stored as Written-more-than-once in another processor's cache, the snooped processor disables a response from main memory and places its own data on the bus, downgrading its status to Valid-and-not-written. The memory's snoop mechanism sees a read cycle which it was not allowed to support, so it grabs a copy of the data as it appears on the bus and temporarily places it into a write buffer in the main memory system itself. As memory latency allows, the new data is written into main memory. In the end, the caches and main memory all have copies of the same data, so the appropriate state for all cached copies of this line is Valid-and-not-written.

Direct data intervention systems which do not use reflection use a different set of cache states which are designed to allow multiple cache copies to

exist of data which has not been updated within main memory. These are known as **ownership** protocols. Assume, once again, that one processor's cache contains a Dirty copy of a main memory location which another processor is requesting. When the cache containing the updated copy snoops this read cycle, it will supply the data to the requesting processor, as in the preceding example, but the main memory will not be equipped with means of updating itself when this transaction occurs. The main memory, in fact, is a traditional design which will only modify itself on system bus write cycles. The only time that main memory is updated in such caches is when the Dirty line is evicted from the cache. This requires a different set of states, since there can now be two Dirty copies of the same location in two different caches. This is where the concept of ownership comes into play.

In ownership protocols, the cache whose CPU modified the Dirty cache location is deemed to be the owner of that location. The owner, and no other cache, is responsible for the eventual update of main memory. Thus, when a Dirty line is replaced in the cache which owns that line, the Dirty line will be evicted from the cache and updated in main memory, whereas the replacement of the same Dirty line in a cache which does not own that line will not cause a main memory update.

So what do we call the case in which no cache owns a copy of a main memory location? This is not widely discussed, but about half of the papers written about ownership give main memory the ownership of any potential cache line which has not been claimed by any cache, while the others say that the line is **unowned**. Main memory ownership of a line balances things out and seems to make it easier to describe the actions taken in an ownership cache coherency protocol, so we will use that nomenclature for the rest of the book. When a cache with this sort of protocol needs to update a line, it will request the line from the owner, whether or not the owner happens to be, in fact, main memory.

One problem addressed by caches using ownership protocols is the difficulty of **pingponging**, which is something like thrashing, in that the state of a line changes frequently at the expense of both the timing on that line and increased bus traffic. Pingponging, besides being a hard word both to spell and to say, is the phenomenon of a line's being taken into and out of a Private state as its value is being interrogated by a cache which does not own it. In an indirect data intervention system, the bus would often be tied up as the reading cache performed abortive reads of the Private location, which the responding cache would answer via memory write cycles, subsequently allowing the reading device back onto the bus. Afterward, the copy which was being modified would again be taken Private, at the expense of a broadcast write, only to be further interrogated by the requesting cache. It wouldn't take too much of this to dramatically tie up a main memory bus.

As an example of how an ownership protocol might work, we'll describe

a typical direct data intervention copy-back cache using four states somewhat similar to those previously used in the indirect data intervention example: Invalid, Valid, Valid-and-written-to-once-by-this-CPU, and Valid-and-written-to-more-than-once-by-this-CPU. "Hey!" you say, "This is almost the same as the last protocol we went through." Well, it is, and it isn't. Bear with us, and you will see the difference that direct data intervention brings to the party.

The last two states are the states which declare that this cache is the owner of the true copy. If we watch a given line through its various state changes, we will see a progression similar to the previous example until a snoop read hit is encountered. Say processors A and B both start out with Invalid lines and get read misses for the same main memory location. The lines will be updated, and their states will be changed from Invalid to Valid. Processor A might then write to this line, and its cache controller, upon seeing a transition in state from Valid to Valid-and-written-to-once-by-this-CPU, will broadcast the write to invalidate other copies on the bus. Let's let processor A write to the line again, raising the status of the line from Valid-and-written-to-once-by-this-CPU to Valid-and-written-to-more-than-once-by-this-CPU.

The copy in processor B's cache was invalidated by the first write, which was broadcast by processor A's cache to the bus, so on the next read miss to this address, the processor will again attempt to update the line. This attempt is now satisfied by the intervening cache from processor A, and the line's state is stored in processor B's cache as Valid, just as if the line had come from main memory. If processor B then replaces this line, no main memory write cycles occur, even though the copy in processor B's cache was more current than the copy contained within main memory. If processor A evicts the line, its cache will update main memory, and processor B's cache will snoop this update, but will not need to change the line's status. Ownership of the line has been relinquished from processor A's cache back to main memory.

Upon careful inspection, the reader will note that the owning processor (processor A) in this example has no idea about whether or not there exist copies of its Valid-and-written-to-more-than-once-by-this-CPU cache line in other processors' caches. To maintain coherency with these copies, the owning processor would have to place every write cycle onto the bus, removing any advantage to using a copy-back protocol. Something needs to be done to allow the cache to know whether write cycles must be broadcast or can remain Private. This requires that the state Valid-and-written-to-by-this-CPU be further divided into two categories: snooped and unsnooped. Unsnooped copies would perform immediate cache write cycles without performing any bus interaction. Snooped copies, knowing that another processor might have a copy of the Dirty location, would need to perform a bus write cycle to invalidate other copies; then the owning cache can take the line's status back to unsnooped. This is where a fifth state of an ownership will come in. We'll rename Valid-and-written-to-by-this-CPU to Valid-and-written-to-by-

this-CPU-but-unsnooped, and make the fifth state be Valid-and-written-to-by-this-CPU-and-snooped.

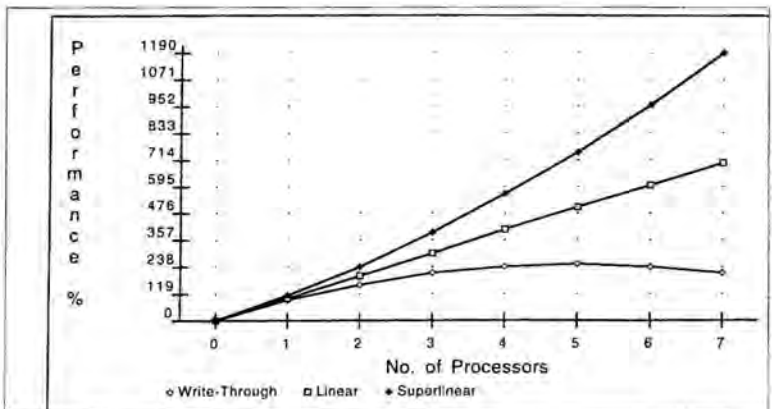
Caches like this one tend to work only with certain bus protocols. The most common at this time (and probably the slowest version we'll discuss) are buses with daisy-chained hierarchies, wherein main memory only responds after all other devices have been given the opportunity to supercede it. If there are several caches tied to the bus, each has its opportunity to respond before main memory is allowed to finally satisfy a read cycle. Split-transaction buses are the next on the list, and, like their use in indirect data intervention systems, they allow the responding cache to give itself a time slot to respond before main memory's access latency, signaling to main memory that the cycle has been satisfied by another respondent. A third method involves the use of an auxiliary bus, which supplies the snooped data while the main memory bus is used exclusively for main memory transfers. The decision of which bus to read then is made on the processor board of the requesting CPU/cache, rather than as a system function. A different set of line states is required to support a two-bus protocol to reroute read cycles, depending upon where the reading processor expects to find the line it needs. Another method, which is used with the  $N+1$  protocol discussed in Section 4.3.4 is the use of a **smart** main memory. A smart main memory keeps a bit for every potential cache line to indicate whether the line is or is not owned by main memory. You will not be surprised when I tell you that designers of such systems call the more common type of design, where the main memory keeps no such status, **dumb** main memories. Smart protocols are a hybrid between cache-based and memory-based protocols, with a leaning more toward the cache-based side.

Both write-once and ownership protocols sometimes take advantage of write allocation as a method of signaling the need for other caches to invalidate any existing copy of a cache line. This means that the only write cycles to the main memory will either be to noncacheable addresses (if they exist) or copy-back cycles, neither of which will cause invalidation, since no other caches contain copies of these lines. Buses used to support such protocols require a mechanism to allow line invalidations during snoop read hits rather than snoop write hits. It will become more clear as we examine such protocols in detail that the bus will be required to support two completely different read cycles, one allowing other caches to maintain existing copies of the requested line, and the other requiring matching lines to be invalidated. Some examples are **Read-for-Ownership** or **Private Read**, **Read Shared** or **Public Read**, **Write-for-Invalidate**, and **Write-without-Invalidation**, and they are described in Section 4.3. Although such states are closely tied to the maintenance of cache coherency, we will treat them as a bus protocol issue and will only describe them in enough depth to allow a thorough understanding of the cache's rather than the bus' operation. As a general

rule, though, Read-for-Ownership cycles are used exclusively for write allocation cycles and not for satisfying read misses.

In sophisticated ownership protocols, however, the compiler instructs the CPU to indicate that a read miss will later result in a write cycle, so the CPU initiates a Read-for-Ownership on read misses to locations which the compiler has decided would best be immediately invalidated in all other caches, locations to which a write is imminent, even though they are brought into the cache via a read miss cycle. Most designs don't have the luxury of being able to have all programs compiled to suit the architecture of the cache, so they count on write misses of naive ownership protocols to generate their Read-for-Ownership cycles.

A really unusual benefit of an ownership coherency protocol is that the cache which owns a copy of a requested piece of data may respond more quickly than main memory, causing there to be certain combinations of hardware and software which will exhibit performance gains which are more than proportional to the number of processors contained within the system (Figure 4.9). Another way to put this is that two processors might perform at 210% of the throughput of a single-processor system, three processors at 325%, and so on. How can this be? Well, as more processors are added, there exist more fast cache locations which can supply data to other processors faster than can main memory, to the point that a large enough system, with just the right programs, would only use main memory as a place to put evicted lines or possibly as a staging area for DMAs. Of course, the benefit of



**Figure 4.9.** Throughput versus number of processors for three kinds of multiprocessing systems. The lower curve shows the effects of main memory bandwidth limitations, the middle line is an idealized linear ratio of performance to number of CPUs, and the top line shows superlinear performance, which can occur in systems using direct data intervention.

this phenomenon is highly dependent on the amount of interprocessor communication which is supported by the software comprising the benchmark.

Some multiprocessor systems use this concept to the hilt, and are sometimes said to cache their own portion of main memory. This is a point at which the delineation between cache and main memory becomes less clear, and what some designers would call cache, others would say is anything but a cache. Kendall Square Research Corporation was one such company with an approach they named Allcache, meaning that the entire main memory was implemented within the caches of several (from 16 to over 1,000) CPUs. One term for systems which have various portions of main memory physically split between all of the different processor boards within a system is **nonuniform memory architectures (NUMA)**. In a nonuniform memory machine, the portion of main memory which is local to the CPU card responds to the local CPU far faster than the other processors, not just because of proximity, but because the local processor need not arbitrate for the system bus and is always given top priority for access to the local memory. From any processor's viewpoint, main memory will have a small fast address range, but all the rest of the address range will be slow. The way to get the best performance out of NUMA machines is to compile code which makes as many references as possible to the local, fast portion of main memory. This implies that the code must be designed to fit the hardware, which is often not an option available to the system design team. Naturally, the converse of a NUMA machine is a **UMA** or **uniform memory architecture** machine, where main memory is equally accessible to all processors, and any one address will respond to all processors with the same latency.

One last piece of bus support for multiple copy-back caches should be mentioned in this section, and that is the **backoff** command. In certain cache designs, backoff is simply a bus command which requires the cache immediately to stop interacting with the system bus. In copy-back cache coherency protocols, the meaning is different, and two types of backoff exist. A **full backoff** is what happens when a snoop hit is not serviced until the snooped cache is able to service the snoop cycle at its own convenience, possibly when the cache is not being used by the CPU. The alternative, a **restricted backoff** is where the snoop hit causes all CPU/cache interaction to come to an immediate halt so that the snoop hit can be instantly serviced. To my knowledge, there is not a large base of thought or research into the merits of one over the other; they are simply viewed as alternative snooping techniques.

### 4.3 EXISTING COPY-BACK COHERENCY PROTOCOLS

This section will show some real examples of existing multiple copy-back coherency protocols. You probably feel by now that this all gets very involved,

so to help you out, we will go through illustrations in the form of tables for each of the protocols. These tables can be very helpful in defining the states in a cache, so try to put one together for your own design, should you do one.

The columns in the table represent the state of the cache at the beginning of any cycle. Each row represents an external even which may have an impact on the way the cache operates or on a change of states within the cache. The tables in this book are not detailed enough to design from, and you will probably find yourself putting together several layers of similar tables to describe the actions of various portions of your own hardware.

### 4.3.1 The MESI Protocol

**MESI** is a write-once cache-based protocol, probably the one most often described in trade magazines. Why not? It has a catchy name. My wife (a student of theology, not engineering) is amused when I tell her that a computer can have a MESI protocol, use SCSI communications, and provide a GUI user interface. It seems somehow inconsistent with all this filthy language, then, that the MESI protocol rules out a cache's use of Dirty bits.

The acronym MESI stands for the four states of a typical indirect data intervention protocol which we described in Section 4.2.5, but uses other, less descriptive, but far simpler names for the same states: **Modified**, **Exclusive**, **Shared**, and **Invalid**. Here is how these four states correspond with those described in Section 4.2.5: Invalid is quite naturally the Invalid state; Shared is the state that we originally called Valid-and-never-written-to-by-any-CPU; Exclusive translates to the Valid-and-written-to-once-by-this-CPU state; and Modified is the name used in the MESI protocol for the state that we originally called Valid-and-written-to-more-than-once-by-this-CPU. As we shall see in the next section, these states are not exclusively applicable to indirect data intervention.

The names make a lot of sense. The Shared state is the only state which allows another copy of the same memory location to be stored within other caches. If one cache has an Exclusive or Modified line, all matching lines in other caches would have been marked Invalid (we'll go through this in more detail in Table 4.1). The Exclusive state signifies to the cache controller that the main memory location is current with the contents of the cache and that no other cache copies of the same main memory location exist; this is an exclusive cache copy of the main memory location (some refer to the Shared state as a **Nonexclusive** state). Finally, the Modified state indicates that the only current version of the address resides within this cache.

The MESI protocol was first developed to be used on the original version of multibus, a bus which does not already have any hooks to support coherency protocols. As a result, there is no write allocations or direct data in-

Table 4.1. MESI

|                        | Invalid   | Shared  | Exclusive   | Modified  |
|------------------------|---|---|---|---|
| <b>From CPU Bus</b>    |   |   |   |   |
| Read Miss              | Update line from main memory. Update status to Shared.                | Update line from main memory. No status change.                       | Update line from main memory. Update status to Shared.      | Evict modified line to main memory. Update line from main memory. Update status to Shared.                |
| Read Hit               | Does not occur.   | Read cache data. No status change.                                    | Read cache data. No status change.                          | Read cache data. No status change.  |
| Write Miss             | Write to cache line. Write to system bus. Update status to Exclusive. | Write to cache line. Write to system bus. Update status to Exclusive. | Write to cache line. Write to system bus. No status change. | Evict modified line to main memory. Write to cache line. Write to system bus. Update status to Exclusive. |
| Write Hit              | Does not occur.   | Write to cache line. Write to system bus. Update status to Exclusive. | Write to cache line. Update status to Modified.             | Write to cache line. No status change.  |
| <b>From System Bus</b> |   |   |   |   |
| Read Miss              | No response. No status change.  | No response. No status change.  | No response. No status change.                              | No response. No status change.  |
| Read Hit               | Does not occur.   | No response. No status change.  | Update status to Shared.                                    | Abort snooped cycle. Write modified line to main memory. Update status to Shared.                         |
| Write Miss             | No response. No status change.  | No response. No status change.  | No response. No status change.                              | No response. No status change.  |
| Write Hit              | Does not occur.   | Update status to Invalid.   | Update status to Invalid.                                   | Update status to Invalid.   |

tervention. All write cycles in an implementation of a MESI system on a simple bus are considered writes-for-invalidate, and will automatically invalidate any matching locations in other caches. This is shown in Table 4.1. Since the bus in such a system is not expected to support a Read-for-Invalidate state, write allocation is not used in the design of MESI caches.

Table 4.1 shows the four MESI states and the responses given in the event of processor and snoop read and write hits and misses in cacheable spaces. Most CPU cycles perform as would similar cycles in either a write-through or a copy-back cache, depending on the state the cache line was in before the transaction occurred. The two special cases are the write miss to a Mod-

ified line and a write hit to a line marked Exclusive. A write miss to a Modified line can either cause a main memory write cycle without updating the cache, or, as is shown in Table 4.1, the current contents of the line can be evicted followed by a write-through cycle which takes the line to the Exclusive state. A write hit to an Exclusive line will update the line to a Modified state without a bus transaction. Exclusive must be entered before going to Modified. No cache line is allowed to go directly from either of the other states to the Modified state.

There are always special cases. Intel's processors implement the MESI protocol using write allocation, to match the disparity between the processors' 16-byte internal cache line size and the processors' support of write cycles of between 1 and 4 bytes. This means that all write misses will appear as read misses on the processor bus. With this being the case, it helps performance if all read misses are first assumed by the system to be write allocation misses, and for the state of a newly fetched cache line automatically to be set to Exclusive, with all other cached copies to be automatically invalidated. This requires the cache/CPU combination to be supported with a main memory bus offering a Read-for-Invalidate mechanism, so that read cycles can cause invalidations in other caches, unless the designer is willing to suffer along with an occasional problem of pingponging, where two caches which share a piece of data continually invalidate the other's copy.

Snoop cycles in the MESI protocol are treated as you would pretty much expect a copy-back cache to perform on snoops. Snoop read and write misses are ignored, and read hits are only really noticed if the snooped cache line is either 1) in an Exclusive state, in which case, the exclusivity will be downgraded to the Shared state (a valid copy already exists in main memory), or 2) in the Modified state, where the other device's read request will need to be aborted, and the snooped cache must then be allowed to update main memory with the contents of the Dirty line before allowing the snooping processor to retry the aborted bus cycle. In either case, the line's status will be changed to the Shared state. All write snoop hits will invalidate the hit cache line, but a write hit to a Modified line is usually an indication of some very bad housekeeping on behalf of the software!

True minimalists will wonder if the Exclusive state can be done away with, and it can, but at a cost. The cache controller can easily enough determine that the cache is transitioning from a Shared state to a Modified state, and send out a bus write to invalidate any other cached copy, just as it would going from the Shared to the Exclusive state, but if the line were never again written to, the cache controller would not know and would evict the Modified line, whether or not it was still coherent with its main memory counterpart. This is not in keeping with the goal of reduced bus traffic, so it is rarely used, but a version of this will be discussed in Section 4.3.4.

### 4.3.2 Futurebus+

An open standard bus protocol, the IEEE Futurebus+ (the plus sign reminds me of a small letter t, but I doubt that it's any indication that this bus will meet with limited acceptance), takes advantage of a wealth of understanding and growth that bus experts have amassed since other, better established, open-standard buses were first conceived. Futurebus+ uses a split-transaction synchronous bus with its own voltage I/O levels called bus-transceiver logic (BTL) which, although they are incompatible with standard logic levels like ECL and TTL, make up for any difference by being blazingly fast.

Futurebus+ specifies a copy-back coherency protocol based on MESI, but far more bus support than the original MESI model is assumed, write allocation is used, and an additional mechanism is included which was given the hysterical name **snarf** or **write snarf**. When a processor is attempting to read data into a cache, but the bus is unavailable, the cache is capable of listening to the other transactions going on, and, if the requested address happens to be involved in a bus transaction, that processor's cache controller will grab a copy and will stop trying to arbitrate for control of the bus. The Futurebus+ specification is written to allow the designer to select from a broad menu of options, and an option exists which allows a cache to snarf a copy of a bus transaction whose address matches the address of an Invalid location, even if the processor is not requesting a copy of it. This looks a lot like reflection, doesn't it? It's almost zenlike in the way that the Futurebus+ committee decided to mimic the main memory technique of reflection, so that caches now capture data from a cache-to-main memory transfer, just as a reflecting memory captures data on a cache-to-cache transfer. It also makes a lot of sense to put reflection onto a processor as well as onto a memory card.

Some readers may disagree, and that's surely because you've considered that the odds of the exact address which the processor is requesting being on the bus while the processor is attempting to gain control of the bus are almost nil, and that the cache should be nearly completely filled with Valid locations. The counterarguments are twofold. First, any protocol which can reduce bus traffic will further increase the number of processors which can be tied to a bus, and, as the bus traffic is reduced, the likelihood of such interactions becomes a larger share of the overall chain of events. Second, those cache locations which are marked Invalid after the processor has been running long enough to fill its cache probably were Valid at one point, then got invalidated by another processor's taking the line into an Exclusive state. If the address again appears on the bus, this automatically implies that the line is being downgraded from its Exclusive state and is again ready to be replicated within other caches in the system.

Just because you use Futurebus+, you are not required to have copy-back

caches on every processor board, nor even to have multiple processors, but since the bus contains all the protocols to support the most extensive implementations, it is straightforward to implement a mix-and-match system of write-through, copy-back, and noncache processor boards. As I said before, there is a vast menu of ways to implement your own board and still be assured of compatibility.

Futurebus+ differs from the MESI protocol upon which it is based in that it uses direct data intervention (the reading of cacheable locations from other caches, rather than from main memory) and write allocation, both of which are supported by a split-transaction bus protocol which has been specified around the cache implementation, rather than the other way around. There are three kinds of read cycles, two writes, and an invalidate, which is a faster way to invalidate a replica of a cache line in other caches than would be a Read-for-Invalidate or Write-for-Invalidate. Further, snoop hit acknowledgments are broadcast from the snooped caches onto the bus via a signal called  $tf^*$ . This  $tf^*$  or transaction flag is a bus signal whose definition depends upon the type of transaction which is occurring on the bus. For all of the cycles in this section,  $tf^*$  signals a snoop hit. By monitoring the  $tf^*$  signal, the requesting cache has the option of taking a line immediately into an Exclusive state, if that line is copied in no other cache. This makes Table 4.2 a bit more complex than Table 4.1 was for MESI, but it is worthwhile in keeping unnecessary write-once cycles off the main memory bus. Reflection is also required of the memory subsystems (if they exist), another bandwidth-saving measure, since high-speed direct data intervention cycles can be used without requiring followup cycles to update the main memory.

Wait a minute! What is this "if they exist" stuff about memory subsystems? Futurebus+ is defined in such a way as to allow the entire system to exist without a main memory, as long as any active main memory address is always accounted for in the system. For this they have the term **repository of last resort**, which is the place where that address eventually is brought to rest. In systems with main memories, this is the owner of the line, which is, more often than not, main memory. In a cache-only system, the repository of last resort is the current owner of the line, and the line is assured, through other parts of the bus protocol which we won't discuss here, to always be replicated somewhere within the system.

To understand the Futurebus+ bus commands using Table 4.2, the names of the four states should be defined. They are Exclusive Modified, which corresponds to the Modified state of the MESI protocol, Exclusive Unmodified (MESI's Exclusive), Shared Unmodified (Shared), and Invalid.

Two of the simplest bus commands to understand are those used by I/O bus masters, normally DMA devices. These are Read Invalid and Write Invalid. Although the Write Invalid command automatically invalidates all

Table 4.2. Futurebus+

|                        | Invalid  | Shared<br>Unmodified  | Exclusive<br>Unmodified  | Exclusive<br>Modified   |
|------------------------|--|---|--|---|
| <b>From CPU Bus</b>    |  |   |  |   |
| Read Miss              | Update line from owner using Read Shared. Update status to Shared. Unmodified if another cache has asserted tf*, else may update status to Exclusive Unmodified. | Update line from owner using Read Shared. No status change if another cache has asserted tf*, else may update status to Exclusive Unmodified. | Update line from owner using Read Shared. Update status to Shared. Unmodified if another cache has asserted tf*, else no status change required. | Evict owned line to main memory using Copyback. Update line from owner using Read Shared. Update status to Shared Unmodified if another cache has asserted tf*, else may update status to Exclusive Unmodified. |
| Read Hit               | Does not occur.  | Read cache data. No status change.  | Read cache data. No status change.   | Read cache data. No status change.  |
| Write Miss             | Update line from owner using Read Modified. Merge write data with new line. Update status to Exclusive Modified.   | Update line from owner using Read Modified. Merge write data with new line. Update status to Exclusive Modified.                              | Update line from owner using Read Modified. Merge write data with new line. Update status to Exclusive Modified.                                 | Evict owned line to main memory using Copyback. Update line from owner using Read Modified. Merge write data with new line. No status change.   |
| Write Hit              | Does not occur.  | Write to cache line. Issue Invalidate to system bus. Update status to Exclusive Modified.   | Write to cache line. Update status to Exclusive Modified.  | Write to cache line. No status change.  |
| <b>From System Bus</b> |  |   |  |   |
| Read Miss              | No response. No status change.   | No response. No status change.  | No response. No status change.   | No response. No status change.  |
| Read Shared Hit        | Does not occur.  | Assert tf*. No status change.   | Assert tf*. Update status to Shared Unmodified.  | Disable main memory response. Output requested data. Assert tf*. Update status to Shared Unmodified.  |
| Read Invalid Hit       | Does not occur.  | Assert tf*. No status change.   | Assert tf*. Update status to Shared Unmodified.  | Disable main memory response. Output requested data. Assert tf*. Update status to Shared Unmodified.  |

Table 4.2. (continued)

|                   |                                |  |  |  |
|-------------------|--------------------------------|--|--|--|
| Read Modified Hit | Does not occur.                | No bus response. Update status to Invalid. | No bus response. Update status to Invalid. | Disable main memory response. Output requested data. Update status to Invalid. |
| Write Miss        | No response. No status change. | No response. No status change.             | No response. No status change.             | No response. No status change.   |
| Write Invalid Hit | Does not occur.                | Update status to Invalid.                  | Update status to Invalid.                  | Update status to Invalid.  |
| Invalidate Hit    | Does not occur.                | Update status to Invalid.                  | Does not occur.                            | Does not occur.  |
| Copyback Hit      | Does not occur.                | Does not occur.                            | Does not occur.                            | Does not occur.  |

cached copies in existence, the Read Invalid command does not, but rather downgrades copies to the Shared Unmodified state, and causes owners of Exclusive Modified copies to intervene in the case of a snoop read hit.

The Read Shared command is issued by a processor which has experienced a read miss cycle, and any snoop hit allows the data to come from either the main memory (if the snooped copy was Shared Unmodified or Exclusive Unmodified) or from the snooped cache in the event that the snoop hit was to an Exclusive Modified location. In all of these cases, the snooped cache will assert  $t^*$  and will come to rest in the Shared Unmodified state. (Being flexible, the Futurebus+ specification does allow a cache to invalidate its own copy of a line if it does not desire to assert  $t^*$ .) Since reflection is used in the main memory, the downgrade from Exclusive Modified to Shared causes no coherency problems. In every case, a valid copy of the line resides in main memory at the end of any Read Shared cycle. Now the requesting cache knows whether other cached copies of the line exist, since it will have received an asserted  $t^*$  if there were any other cached copies of the line, and a negated  $t^*$  if there were not. This being the case, the new cache line will be loaded as Shared Unmodified if there exist other cached copies, and will be immediately loaded into the Exclusive Unmodified state if no other cache has a copy. If the line which is loaded as Exclusive Unmodified is then the subject of a cache write hit, no write-first cycle needs to be invoked, thereby saving bus bandwidth over a standard MESI implementation.

The final Futurebus+ read cycle is a Read Modified, which is used by a cache to signal that it has suffered a write miss cycle, and must obtain an Exclusive copy of a line for its write allocation. When a Read Modified snoop hit occurs on a cache, that cache's copy of the line is invalidated. Since the line might be longer than the word which is being written (Futurebus+'s line length is 64 bytes), and since snooped Exclusive Modified copy may be modified in different bytes than would be modified by the requesting processor,

the owner of a snooped Exclusive Modified line must perform a direct data intervention to hand the data to the new owner.

The two Futurebus+ write cycles are Write Invalidate, which has already been discussed, and Copyback. The Copyback cycle is used during evictions of Exclusive Modified lines. Since these lines are Exclusive, there can exist no replicas of them in other caches, so there will never be any snoop hits.

The only other Futurebus+ bus cycle which involves cache interaction is the Invalidate cycle. Invalidate is issued by a cache which owns a Shared copy of a line and wants to take that line into an Exclusive Modified state. Strangely enough, the Futurebus+ committee decided to call this a write miss cycle. In the name of consistency, I will follow a more general convention and call this a write hit to a Shared Unmodified location. Upon a snoop hit of an Invalidate command, any other cache which owns a Shared Unmodified copy of the line will have its copy Invalidated. It is impossible for a copy in another cache to have any other state, since a line which is in a Shared Unmodified state in one cache cannot be in either of the Exclusive states in any other cache.

In taking this relatively backward approach, we have discussed all of the snoop cycles at the bottom of Table 4.2 and have skipped the CPU cycles. Let's go over them. On a CPU read miss, if the line to be replaced is in an Exclusive Modified state, it must be evicted before the new data is read into the cache. This is done by writing the evicted line to main memory using the Copyback bus command. Then the same series of events is taken as during a CPU read miss to a line in any of the other states. A Read Shared command is placed on the bus, and the owner of the data responds, plus any snooped caches are given an opportunity to assert  $tf^*$  to tell the requesting cache that the line is indeed Shared. The line is copied into the cache, and, if  $tf^*$  is asserted, the line is marked Shared Unmodified, but, if  $tf^*$  is not asserted, the line goes immediately to the Exclusive Unmodified state. Of course, the requesting cache does not absolutely have to put the line into an Exclusive Unmodified state, as Futurebus+ allows this as an option, but it's an option which is sure to add speed to the overall system's performance since it removes the need to perform a subsequent write-once cycle. Naturally, on read hits, the data goes straight from the cache to the CPU without any bus interaction or state changes.

On a write hit, an Exclusive Modified line will be written to without bus interaction or state change, while an Exclusive Unmodified line will be written to without bus interaction, but will undergo a state upgrade to Exclusive Modified. If the line started out in a Shared Unmodified state, other copies must be Invalidated, so the cache controller must issue an Invalidate command on the bus. This is a fast, abbreviated version of a write cycle, where data is never placed on the bus. The cache line can then be written to, and its status will be upgraded to the Exclusive Modified state.

Write miss cycles, like read misses, are all satisfied the same way, once any possible dirty line has been evicted. The eviction is handled the same as for a read miss: If the line is in an Exclusive Modified state, it is first written to main memory using the Copyback bus command. Then, for any line status, the line to be written to is read from its owner into the cache using the Read Modified command, invalidating all other cached copies, is written into, and is immediately put into the Exclusive Modified state. The status of  $tf^*$  is ignored since the Read Modified command will invalidate any snooped copies.

Another unique facet of Futurebus+ is that it allows coherency to be maintained on hierarchical bus structures like the one shown in Figure 4.10. The key to this mind-bending problem is the use of two sorts of agents: **memory agents** and **cache agents**. A memory agent will receive read and write commands on one side of the bus and will respond to those commands with transactions which appear on that side of the bridge to be coming from

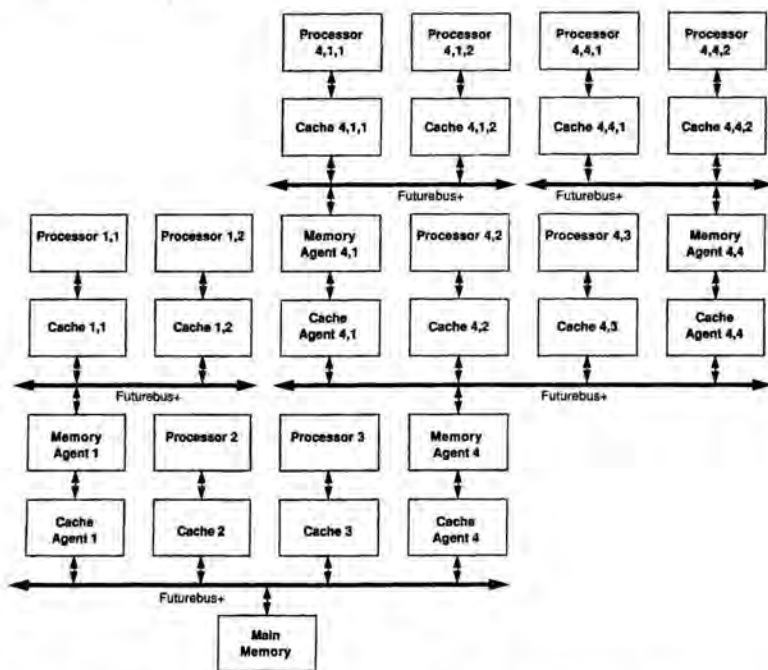


Figure 4.10. A large Futurebus+ system where five levels of the Futurebus+ are used. Coherency is maintained between all caches through the actions of the cache agents and the memory agents.

nothing but a simple main memory on the other side of the bridge. On the other side, the memory agent keeps track of caches and main memory locations, monitors the ownership of each line, and transacts with each line's owner on an individual basis. Since the memory agent must know whether to go to main memory or to a cache to get the current value of a piece of data, caches on the memory agent's side of the bridge cannot be allowed to use the Exclusive Unmodified state. The Exclusive Unmodified state allows a line to go into an Exclusive Modified state without bus interaction, so the memory agent would not discover when the ownership of the line transferred from the memory on that bus to the cache. If you look again at Table 4.2, you can see that the Exclusive Unmodified state is only entered if a read miss is serviced without the  $t^*$  line going active, and then only if the designer wishes to implement this. Otherwise, a read miss takes the new line into the Shared Unmodified state. If the cache resides on a side of a bridge which is monitored by a memory agent, then the response to the  $t^*$  on a memory read miss is simply disallowed.

A cache agent has only caches to worry about, so it is a bit simpler. Based on inclusion, the cache agent snoops the bus on the other side of the bridge from the side used by the caches it represents. The cache agent knows the addresses of every line which is cached on its own bus, and allows through to its own side of the bridge only those snoop cycles which are likely to cause snoop hits in the caches which it protects. This is a way of cutting down the bus traffic which flows across the bridge. By using cache agents and memory agents, snoops can be propagated up and down the hierarchy of Figure 4.10 without tying all the buses down in massive quantities of fruitless and irrelevant snoop traffic.

Another new multiple-processor bus specification is the Corollary C-bus II. This specification is very much like Futurebus+ since it uses a modified version of the MESI protocol with the additional support of write allocation, 32-byte lines, and direct data intervention. To support all of this, two special read-exclusive bus cycles are provided: Read Exclusive and Evict Exclusive. Both are used to support write allocation, with the Evict Exclusive command preceding its write allocation read cycle with the eviction of a Dirty line. Snooped hit copies of the line read into the cache using either of these cycles are invalidated by either of these commands in all other caches.

### 4.3.3 The MOESI Protocol

Although the **MOESI** acronym looks almost the same as MESI, it represents a very different protocol since it supports direct data intervention. The acronym stands for states with the same names as used in MESI, with the addition of the **Owned** state, but the meanings of the states are mostly differ-

ent, and correspond to the long-named states used in Section 4.2.5 in the following manner: Invalid once again is the Invalid state; Shared likewise is the state originally called Valid; Exclusive, however, becomes Valid-and-written-to-once-by-this-CPU; Modified now becomes Valid-and-written-to-more-than-once-by-this-CPU-but-unsnooped; and Owned is added to mean Valid-and-written-to-more-than-once-by-this-CPU-and-snooped. The Modified, Owned, and Shared states show the difference between MESI and MOESI. If a processor in MESI has a Shared line, it means that main memory is current and that the line has not been written to in any cache. In the MOESI protocol, Shared means that this is just a copy of a location which may or may not be current in main memory, but is an exact copy of the owner's data. In MESI, the Modified state implicitly means that the location has not been snooped, whereas this needs to be spelled out in MOESI. Finally, the Owned state accounts for the difference between the direct data intervention of MOESI and the indirect data intervention supported by MESI. Owned locations may have replicas in other caches, similar to Shared lines, but main memory has never been updated, so it must be written to by the owning cache upon an eviction.

Table 4.3 shows the five MOESI states, and the responses given in the event of processor and snoop read and write hits and misses in cacheable spaces. Like MESI, the MOESI model assumes no special bus support. State transitions are described in detail in the following paragraphs.

On a CPU read miss, if the target line is Owned or Modified, it is evicted. The cache then updates the line either from main memory or from the owning cache, and the line will be loaded as Shared. If the line comes from an owning cache, that implies that this cached copy is either in the Owned or Modified state, and, if the state is Modified, it will be downgraded to Owned. During snoop read hits to Shared and Exclusive copies of a line, there will be no bus interaction, but, if an Exclusive copy receives a snoop hit, it will downgrade its own status to Shared.

As in the MESI protocol, all main memory write cycles are snooped for invalidation by all other caches. MESI and MOESI designs tend not to use write allocation, meaning that a CPU write miss must broadcast itself through the system as a bus write cycle. The write cycle then will be transmitted through the cache similarly to a write-through cache. In many designs, write misses are ignored by the cache. The assumption used to justify the cache's ignoring write misses is that anything which is not already in the cache when it is written is very possibly not going to be referred to in the future. This is indeed the case for such items as initialization values loaded into various areas, but is not the case for the first time a memory resident loop counter is set up, nor for the first time the stack reaches a certain depth. Still, these are the exception and not the rule, so write misses are usually ignored in MOESI caches.

Table 4.3. MOESI

|                        | Invalid   | Shared  | Exclusive   | Owned  | Modified  |
|------------------------|---|---|---|--|---|
| <b>From CPU Bus</b>    |   |   |   |  |   |
| Read Miss              | Update line from main memory. Update status to Shared.                | Update line from main memory. No status change.                       | Update line from main memory. Update status to Shared.      | Evict Owned line to main memory. Update line from main memory. Update status to Shared.                | Evict Modified line to main memory. Update line from main memory. Update status to Shared.                |
| Read Hit               | Does not occur.   | Read cache data. No status change.                                    | Read cache data. No status change.                          | Read cache data. No status change.   | Read cache data. No status change.  |
| Write Miss             | Write to cache line. Write to system bus. Update status to Exclusive. | Write to cache line. Write to system bus. Update status to Exclusive. | Write to cache line. Write to system bus. No status change. | Evict Owned line to main memory. Write to cache line. Write to system bus. Update status to Exclusive. | Evict Modified line to main memory. Write to cache line. Write to system bus. Update status to Exclusive. |
| Write Hit              | Does not occur.   | Write to cache line. Write to system bus. Update status to Exclusive. | Write to cache line. Update status to Modified.             | Write to cache line. Write to system bus. Update status to Exclusive.                                  | Write to cache line. No status change.  |
| <b>From System Bus</b> |   |   |   |  |   |
| Read Miss              | No response. No status change.  | No response. No status change.  | No response. No status change.                              | No response. No status change.   | No response. No status change.  |
| Read Hit               | Does not occur.   | No response. No status change.  | No response. Update status to Shared.                       | Disable main memory response. Output requested data. No status change.                                 | Disable main memory response. Output requested data. Update status to Owned.                              |
| Write Miss             | No response. No status change.  | No response. No status change.  | No response. No status change.                              | No response. No status change.   | No response. No status change.  |
| Write Hit              | Does not occur.   | Update status to Invalid.   | Update status to Invalid.                                   | Update status to Invalid.  | Update status to Invalid.   |

In the example in Table 4.3, we have chosen to take the more complex route of updating the missed line. This can only be done if the cache line is the same size as the smallest write cycle supported by the CPU/software of the system (not all software uses byte write cycles when the CPU provides them). If smaller writes are allowed, and if a write miss is to be written into the cache, write allocation protocols should be considered. The table assumes that only word writes are supported by the cache. Since the design requires the replacement of a Modified line on a write miss, the bus subsequently must carry two cycles, an eviction followed by a write (for invalidation) of the replacement line.

Should write allocation be used, one of the several three-state and four-state protocols described in the following sections would probably be chosen over the five-state MOESI. In a MOESI system with write allocation, write misses to a Modified location would consume three bus cycles: first the eviction of the Modified line, then the read cycle to allow the allocation, then a write-once cycle to invalidate any other cached copies of the line. This can be a pretty heavy penalty!

On a CPU write hit, if the location is Modified, there is no bus activity, and the line is updated, remaining in the Modified state. Similarly, a CPU write hit to a line marked Exclusive will update the line without invoking any bus transactions, and the resultant line will end up in the Modified state. If the line is Shared or Owned, the line is updated, a write cycle to the bus is initiated to invalidate any other copies which may exist in other caches, and the state is updated to Exclusive. This is kind of weird when an Owned line is written to, since what appears to be a higher status, a Written-to-more-than-once status, is downgraded to Exclusive, a Written-to-once status, in response to a write hit. The rationale behind this is that an Owned line might be copied in another CPU's cache, so a bus cycle is absolutely necessary to invalidate any copies which could be stale in other caches. Since the main memory bus is assumed not to differentiate between different kinds of write cycles, an invalidation, rather than an update, must occur on a write snoop hit to assure that any cache wishing to take its copy to an Exclusive state through a main memory write cycle can do exactly that. Once this bus cycle has happened, the main memory is current, and there is no longer any need for the cache with the new Exclusive line to evict that line. Since both the Owned and Modified states require an eviction on a line replacement, they are inappropriate, and that leaves us with Exclusive as the only rational destination after a CPU write hit to an Owned line. Naturally, there is no such thing as a write hit to an Invalid location.

A write snoop hit always results in the snooped line being immediately invalidated, no matter what the original status was. This can signify software problems, just as it did in the MESI cache, if a dirty or Owned line gets invalidated. It might seem a waste that the eviction of an Owned copy will cause in-

validation of coherent Shared copies of the same line in other caches. These caches will have to go back to main memory after the eviction to get the same data they had before. Further, this process will happen on every write hit cycle to an Owned location. This might rule out the use of MOESI in favor of other protocols in the design. The one beauty of MOESI which would offset this is that the alternatives require special bus architectures, whereas all MOESI needs is a prioritizing scheme to allow caches to turn off main memory in the event of a snoop write hit. This is quite a significant consideration in designs where the design team must fit a multiple-processor coherency protocol to an existing bus which was designed without coherency in mind.

On snoop read hits, Modified lines will supply their data directly to the requesting cache and will downgrade their status to Owned. Owned lines will respond to a snoop read hit in the same way, without any change to the responding cache's line status. Read snoop hits to Exclusive and Shared Locations cause no data to be supplied from the cache to the bus, since the data in that cache should be supplied by the owner, whether the owner is main memory, which is always the case with an Exclusive line but may or may not be the case with a Shared line. The owner may be a different cache. The Exclusive copy will, however, downgrade its status to Shared. If the snooped read hit is to an Owned or Modified line, data will be supplied by the cache to the bus, and the final state of the line will be Owned. Nope, there are no snoop read hits to Invalid lines.

The attraction to the MOESI protocol over MESI is that it uses direct data intervention. This speeds up interprocessor communication at a slight cost in cache complexity, since the cache must now track five states per line, rather than four. Either protocol uses standard main memory write cycles to invalidate matching locations in other caches, and neither uses write allocation, which is more complex, but actually helps to simplify some of the protocols to be discussed in the following sections.

#### 4.3.4 N+1

A different protocol was developed by Synapse for their N+1 fault-tolerant computer (Table 4.4). This protocol uses a smart main memory (described in Section 4.2.5) with a single bit called a usage-mode bit maintained in main memory for each potential cache line, to indicate whether that line is owned by the main memory or by a cache. The usage-mode bit tells whether the line is Public or Private. This extra bit removes the need for caches to preempt main memory from responding to a system bus cycle since the main memory will disable itself from responding if the usage-mode bit is set, saving some time over daisy-chained or other less costly memory-disabling schemes. The main memory in the N+1 also accelerated its own performance through the use of a 15-entry queue on all memory boards.

Table 4.4. N+i

|                        | Invalid  | Valid  | Dirty   |
|------------------------|--|--|---|
| <b>From CPU Bus</b>    |  |  |   |
| Read Miss              | Update line from main memory using Public Read.<br>Update status to Valid.   | Update line from main memory using Public Read.<br>No status change.   | Evict dirty line to main memory.<br>Reset main memory Usage-Mode bit.<br>Update line from main memory using Public Read.<br>Update status to Valid.   |
| Read Hit               | Does not occur.<br>No status change.   | Read cache data.<br>No status change.  | Read cache data.<br>No status change.   |
| Write Miss             | Update line from main memory using Private Read.<br>Set main memory Usage-Mode bit.<br>Write to cache line.<br>Update status to Dirty. | Update line from main memory using Private Read.<br>Set main memory Usage-Mode bit.<br>Write to cache line.<br>Update status to Dirty. | Evict dirty line to main memory.<br>Reset main memory Usage-Mode bit.<br>Update line from main memory using Private Read.<br>Set main memory Usage-Mode bit.<br>Write to cache line.<br>No status change. |
| Write hit              | Does not occur.  | Update line from main memory using Private Read.<br>Set main memory Usage-Mode bit.<br>Write to cache line.<br>Update status to Dirty. | Write to cache line.<br>No status change.   |
| <b>From System Bus</b> |  |  |   |
| Read Miss              | No response.<br>No status change.  | No response.<br>No status change.  | No response.<br>No status change.   |
| Public Read Hit        | Does not occur.  | No response.<br>No status change.  | Abort snooped cycle.<br>Write dirty line to main memory.<br>Reset main memory Usage-Mode bit.<br>Update stats to Invalid.   |
| Private Read Hit       | Does not occur.  | Update status to Invalid.  | Abort snooped cycle.<br>Write modified line to main memory.<br>Reset main memory Usage-Mode bit.<br>Update status to Invalid.   |
| Write Miss             | No response.<br>No status change.  | No response.<br>No status change.  | No response.<br>No status change.   |
| Write Hit              | Does not occur.  | Does not occur.  | Does not occur.   |

Each CPU card in the N+1 contained a 16K-byte copy-back physical cache, each line of which had two status bits: Valid and Data-Modified. The Valid bit was used to assert the validity of the cache line (naturally), while the Data-Modified bit kept track of the coherency of the data in main memory. Synapse claimed to be able to support up to 28 processors on a system, each adding incrementally to the system's throughput.

This protocol only used three states: Invalid, Valid, and Dirty. The three states were effectively used to accomplish the same thing as other four-state coherency protocols (some call these states Invalid, Shared, and Modified, or **MSI**). Support for the three-state methodology came in two forms. First, write allocation was used for all write miss cycles as well as write hit cycles to Valid locations (write hits to Dirty locations required no bus interaction), so that the only bus writes to cacheable addresses were the result of evictions or read snoop hits. Second, the bus allowed two sorts of read cycles: Public Read and Private Read. Public Read cycles were used exclusively for read miss cycle line updates, where the main memory usage-mode bit remained reset (indicating that main memory continued to own the cache line), and other Valid copies were allowed to coexist in other caches. Private Read cycles were incurred for all write allocation cycles, those line update read cycles which were incurred specifically in support of write miss cycles or writes to Valid locations which were being taken to the Dirty state.

Something somewhat bizarre about this protocol is that a write hit to a location which was not already Dirty caused the cache to reread the location from main memory using a Private Read command. The line would have previously been read for a read miss with a Public Read command. The repeated read cycle both invalidated any other copies of the cache line before the missed write cycle occurs and set the main memory's usage-mode bit. The newly fetched line was loaded into the cache as Dirty. What is odd is that this process was not optimized for the way that most code performs write cycles. Writes are most often performed to locations which have been previously read. For the N+1 protocol, this meant that most write cycles caused two main memory read cycles: a Public Read cycle for the reading of the data and a Private Read cycle for the write. This was no larger a number of cycles than is required by MESI, since MESI performs a main memory read first, then the first write, but the N+1 caused more bus traffic than some of the following protocols which involve less main memory cycles.

In a step-by-step analysis of N+1, we see the familiar fact that all CPU read hits are serviced immediately from the cache, without bus interaction or state change. On a CPU read miss, if the target line is Dirty, it will be evicted, and in all cases a Public Read bus command will get the updated line from main memory. The new line is stored as Valid.

On a CPU write hit, if the line is Dirty, the data is overwritten without status change or bus interaction. If the line is Valid, however, it is reacquired

from main memory, this time with a Private Read cycle, which will signify to snooped caches that they should invalidate their own copies of the line, and will set the main memory's usage-mode bit. The local cache will immediately place the line into a Dirty state, and the new data will overwrite the appropriate portion of the line. A similar cycle is used to support a CPU write miss cycle, since the N+1 protocol is based around write allocation. If the missed line is Dirty, it will be evicted. No matter what the state of the existing line, a Private Read will be used to perform the write allocation, and the line will come to rest in the Dirty state with the new data overwriting what was supplied by main memory. All Private Read cycles set the usage-mode bit in the main memory. There is one usage-mode bit per potential cache line, and this bit tells the main memory not to respond to a read cycle to that address, deferring to the cache which holds a Dirty copy. This extra bit allows the main memory to respond to read cycles much more quickly than it could if the more typical daisy-chained priority mechanism were used.

Snooping in the N+1 protocol is relatively simple. If the starting state of the snooped line is Invalid, no action will be taken. If the snooped line is marked as Valid, it will only respond to snoop hits which result from Private Reads, in which case, the line will simply be set to Invalid. On a snooped Public Read hit or Private Read hit to a Dirty line, indirect data intervention will be used. This means that the read cycle will be aborted, the cache with a Dirty copy will write its copy back to main memory and will clear the usage-mode bit, the read will be allowed to be retried, and the Dirty line will be invalidated (I'm not sure why, for a Public Read, the Dirty line won't simply be downgraded to Valid). This is where the balance becomes apparent in how the usage-mode bit is handled. When a Private Read moves a line into a cache, the line is set to Dirty, and the usage-mode bit is set. When the Dirty line is downgraded, main memory is updated, and the usage-mode bit is cleared.

You will notice in Table 4.4 that no allowance is made for the possibility of a write snoop hit, which is kind of odd. How come? Since the N+1 protocol calls for write allocates, all write cycles to Shared or other cache locations are initiated by either a read cycle or an invalidate. This means that the only write activity on the main memory bus will consist of evictions of Dirty locations (which necessarily will not result in snoop hits, since no other cache will contain a copy of a Dirty line) and writes to noncacheable areas. Ipso facto, there is absolutely no reason even to include logic to handle snoop write hits. It appears that this architecture does not use DMAs to handle any disk I/O.

#### 4.3.5 Berkeley

The Berkeley protocol is detailed in Table 4.5. There are four states, as there are in the MESI protocol, named Invalid, Unowned, Owned Non-Exclu-

Table 4.5. Berkeley

|                        | Invalid  | Unowned  | Owned<br>Non-Exclusively   | Owned<br>Exclusively   |
|------------------------|--|--|--|--|
| <b>From CPU Bus</b>    |  |  |  |  |
| Read Miss              | Update line from owner using Conventional Read. Update status to Unowned.  | Update line from owner using Conventional Read. No status change.  | Evict owned line to main memory using Write-without-Invalidation. Update line from owner using Conventional Read. Update status to Unowned.  | Evict owned line to main memory using Write-without-Invalidation. Update line from owner using Conventional Read. Update status to Unowned.                          |
| Read Hit               | Does not occur.  | Read cache data. No status change.   | Read cache data. No status change.   | Read cache data. No status change.   |
| Write Miss             | Update line from owner using Read-for-Ownership. Merge write data with new line. Update status to Owned Exclusively. | Update line from owner using Read-for-Ownership. Merge write data with new line. Update status to Owned Exclusively. | Evict owned line to main memory using Write-without-Invalidation. Update line from owner using Read-for-Ownership. Merge write data with new line. Update status to Owned Exclusively. | Evict owned line to main memory using Write-without-Invalidation. Update line from owner using Read-for-Ownership. Merge write data with new line. No status change. |
| Write Hit              | Does not occur.  | Write to cache line. Write-for-Invalidation to system bus. Update status to Owned Exclusively.                       | Write to cache line. Write-for-Invalidation to system bus. Update status to Owned Exclusively.   | Write to cache line. No status change.   |
| <b>From System Bus</b> |  |  |  |  |
| Read Miss              | No response. No status change.   | No response. No status change.   | No response. No status change.   | No response. No status change.   |
| Conventional Read Hit  | Does not occur.  | No response. No status change.   | Disable main memory response. Output requested data. No status change.   | Disable main memory response. Output requested data. Update status to Owned Non-Exclusively.   |
| Read-for-Ownership Hit | Does not occur.  | No bus response. Update status to Invalid.   | Disable main memory response. Output requested data. Update status to Invalid.   | Disable main memory response. Output requested data. Update status to Invalid.   |
| Write Miss             | No response. No status change.   | No response. No status change.   | No response. No status change.   | No response. No status change.   |

Table 4.5. (continued)

|                                |                 |   |   |   |
|--------------------------------|-----------------|---|---|---|
| Conventional Write Hit         | Does not occur. | No bus response.<br>Update status to Invalid. | No bus response.<br>Update status to Invalid. | No bus response.<br>Update status to Invalid. |
| Write-for-Invalidation Hit     | Does not occur. | No bus response.<br>Update status to Invalid. | No bus response.<br>Update status to Invalid. | Does not occur.                               |
| Write-without-Invalidation Hit | Does not occur. | No response.<br>No status change.             | Does not occur.                               | Does not occur.                               |

sively, and Owned Exclusively. The states are relatively similar to the MESI states, with the exception that the Owned Non-Exclusively state is a Shared Modified state. The MESI Shared state maps closely into the Berkeley Unowned state (so we can assume that the Berkeley crew didn't embrace the idea that main memory is the owner if no cache owned a copy of a line), MESI's Modified state is akin to Berkeley's Owned Exclusively state, and there's never much difference between any two protocols' Invalid states. Invalid is Invalid. But the Owned Non-Exclusively state is a state where the main memory is not coherent with the cached copies of the data, yet more than one cached copy may exist.

Going through Table 4.5, a CPU read miss to a line which is either Invalid or Unowned will simply update the line using a conventional read cycle, but, if the line starts out as either Owned Non-Exclusively or Owned Exclusively, then the line is Dirty and must be evicted. The difference between these two states is that the Owned Non-Exclusively line might be replicated in another cache, even though it is not coherent with main memory. After the eviction, the line is updated the same as for the other two states, and, for all four cycles, the line comes to rest in an Unowned state. CPU read hits, as with the other cache protocols, are simply satisfied from the cache without status change.

The reader will note that the Berkeley protocol uses the terms Conventional Read and Conventional Write instead of the clearer Public Read and Public Write used in other protocols. The meaning is the same; only the terminology is different. Conventional Reads are read cycles owing to CPU read misses, reads to noncacheable addresses, or DMA activity. Conventional Writes only come from DMA activity or noncacheable write cycles.

CPU write misses cause write allocation cycles, where a Read-for-Ownership bus cycle must be used to invalidate any other cached copies of the same line. As we saw in the Futurebus+ and the N+1 protocols, the use of write allocation requires special bus cycles to be used to support some sort

of invalidating read cycle. The replaced line will first need to be evicted if it starts out as either Owned Non-Exclusively or Owned Exclusively, the same as in the case of CPU read misses, but the state of the line after the CPU write miss has been serviced is Owned Exclusively.

CPU write hits are handled differently than the other protocols we have examined. If the line starts the cycle out as Unowned or Owned Non-Exclusively, other copies will have to be invalidated, so the cache sends a Write-for-Invalidation cycle to the bus, just as would happen with MESI or MOESI write-once cycles. A Write-for-Invalidation cycle is a fast write which does not update main memory, but does invalidate matching lines in other caches, just like the Invalidate signal in Futurebus+. The line's state is then changed to Owned Exclusively. If the line started out as Owned Exclusively, then it is simply updated without any bus interaction. If the line started out as Invalid, there can be no CPU write hit cycles.

Other than the special reads and writes, the Berkeley protocol requires the support of two bus mechanisms: write allocation and direct data intervention. As we have seen in other protocols, write allocation requires two sorts of read cycles: Conventional Read and Read-for-Ownership. Read-for-Ownership is issued when a write allocation cycle is in process, that is, the cache is updating a line into which data from a write miss cycle will be merged. Direct data intervention is used to allow the owning cache to supply a replacement line to the requesting cache for either type of read cycle.

Since there are two types of read cycle and three writes, snooping in the Berkeley protocol involves a lot of possibilities. The two reads are Conventional Read, used to replace lines during CPU read misses and for DMA read cycles; and Read-for-Ownership, which is used to support the read portion of write allocation, and must cause invalidation of other cached copies of the requested line. The three write cycles are Conventional Write, used by CPUs to write to noncacheable addresses and by DMA devices; Write-for-Invalidation, which is used to take a line from a nonexclusive state to an exclusive state by invalidating any other cached copies in the system (on a CPU write hit to an Unowned or Owned Non-Exclusively line) and, as was mentioned before, does not require the main memory to record the written data; and Write-without-Invalidation, which is used during evictions to move a Dirty line which needs to be replaced back into main memory.

Snoop hits of Conventional Read cycles only cause responses from lines which are owned by the snooped cache. This means that a response will come from a cache with a copy of the line either in the Owned Non-Exclusively or the Owned Exclusively state. The data is supplied from the cache to the reading device after the cache has disabled main memory from responding, and the final state of the line is Owned Non-Exclusively. Main memory is not updated during these intervention cycles. This allows fast cache-to-cache trans-

fers to occur when necessary, but reduces the number of slow main memory writes to the absolute minimum required. The same process occurs with a snooped Read-for-Invalidation cycle, except that the final status of the line is Invalid, even if the line starts out in the Unowned state.

Snooped Conventional Writes and Write-for-Invalidation cycles both cause a line starting in any state to become Invalid. No data is lost as a result, even though the system supports lines which are longer than the CPU's shortest write cycle, since Write-for-Invalidation cycles will only cause a hit to a line which is exactly copied in the cache which issues the Write-for-Invalidation command. A Write-for-Invalidation snoop hit can only be encountered by lines in either nonexclusive state (Unowned and Owned Non-Exclusively), since they are only issued upon CPU write hits to cache lines in the originating cache. If the line exists, then it cannot exist in an Owned Exclusively state in any other cache. The difference between Conventional Write and Write-for-Invalidation cycles, then, is only in the way that main memory responds to them. The Write-for-Invalidation cycle does not need main memory to update itself, since this write cycle terminates with the line being owned by a cache, so the only write cycle which needs to last long enough for main memory to be able to absorb a copy is the Conventional Write.

A Write-without-Invalidation cycle can only be snooped by a cache with a matching line in the Unowned state, since the issuing cache has the line in either the Owned Non-Exclusively or the Owned Exclusively state. The end result of a snoop hit during a Write-without-Invalidation cycle is simply that the cached line's ownership will move from the owning cache to main memory, so the cache with the Unowned line doesn't really need to care about this transaction. As a result, there is no response to snooped Write-without-Invalidation cycles.

Two of the three write cycles, Write-for-Invalidation and Write-without-Invalidation, are used in cache data transfers. Write-for-Invalidation is used during a write hit cycle to an Unowned or Owned Non-Exclusively line and allows all snooped caches' matching lines to be invalidated as the writing processor takes its own copy of the cache line to the Owned Exclusively state. Write-without-Invalidation is used in the eviction of either an Owned Exclusively or an Owned Non-Exclusively line during replacement for either a read or write miss cycle. The Write-without-Invalidation cycle takes advantage of the fact that the writing cache knows full well that no other cached copies exist, so it doesn't interrupt the other caches with snoop cycles which it knows in advance will have absolutely no effect other than to slow down the other processors. This is important if the Berkeley protocol is implemented in a multiplexed cache-tag RAM design, and the processor is halted every time the cache-tag RAM is to be snooped.

Here's an example of how two Berkeley protocol caches might play

against one another. Say there were only two CPU/cache subsystems on a system built around the Berkeley protocol. The same line could reside in a Valid state in the two caches only as either Unowned/Unowned or as Unowned/Owned Non-Exclusively. In the first case, the main memory would contain the most current version of the cache line, but in the second case, both caches would contain data which was more current than that in the main memory. It would be up to the cache which contained the Owned Non-Exclusively copy to update main memory when that line was replaced.

So, to sum up, a CPU write miss causes a Read-for-Invalidation. A CPU write hit causes a Write-with-Invalidation when the hit line is not in an Owned Exclusively state, and it causes no bus cycle if the line is Owned Exclusively. Read misses cause Conventional Read cycles and are directly intervened, if possible. Evictions are handled via Write-without-Invalidations. Snooped Conventional Read hits to an Owned Exclusively line result in intervention and that line's being downgraded to an Owned Non-Exclusively, but if the line has an Unowned or Owned Non-Exclusively state, data is supplied by main memory and the state is unchanged. Read-for-Invalidation snoop hits cause invalidation. Conventional Write and Write-for-Invalidation snoop hits cause invalidations, and Write-without-Invalidations are ignored if they cause snoop hits to Unowned lines, the only type of snoop hit that such a cycle can invoke. Whew!

#### 4.3.6 University of Illinois

The University of Illinois protocol (Table 4.6) supports direct data intervention similarly to Futurebus+, Berkeley, and the MOESI protocol. Like Berkeley, the N+1, and Futurebus+, this requires more bus support than MOESI, in that the Illinois protocol has a Read-for-Ownership cycle, a conventional Read cycle, and an invalidate cycle, as well as a signal which indicates to the reading cache whether the line which it is reading has been supplied from another cache or from main memory (very much like the  $t_{f}^*$  signal in Futurebus+). Main memory in this system is designed to respond to simultaneous read and write commands as it would to a simple write cycle, since this is what is intended during intervention, which the Illinois protocol supports by placing both cycles simultaneously on the bus. In addition, write allocation is used as it is in the N+1 to remove the need for individual caches to snoop bus write traffic. The four states used in the Illinois protocol are Invalid, Shared, Valid-Exclusive, and Dirty, all of which bear a strong similarity to their counterparts in the MESI protocol. The only state which supports copies of the same line in multiple caches is Shared.

Starting off in an Invalid state, assume we get a CPU read miss cycle. The cache issues a standard Read cycle to all other caches and the main memory.

Table 4.6. University of Illinois

|                        | Invalid  | Shared  | Valid-Exclusively  | Dirty  |
|------------------------|--|---|--|--|
| <b>From CPU Bus</b>    |  |   |  |  |
| Read Miss              | Update line from owner.<br>If from another cache, update status to Shared.<br>If from main memory, update status to Valid-Exclusive. | Update line from owner.<br>If from another cache, no status change.<br>If from main memory, update status to Valid-Exclusive. | Update line from owner.<br>If from another cache, update status to Shared.<br>If from main memory, no status change. | Evict dirty line to main memory.<br>Update line from owner.<br>If from another cache, update status to Shared.<br>If from main memory, update status to Valid-Exclusive. |
| Read Hit               | Does not occur.  | Read cache data.<br>No status change.   | Read cache data.<br>No status change.  | Read cache data.<br>No status change.  |
| Write Miss             | Update line from owner, using read-for-ownership.<br>Write to cache line.<br>Update status to Dirty.                                 | Update line from owner, using read-for-ownership.<br>Write to cache line.<br>Update status to Dirty.                          | Update line from owner, using read-for-ownership.<br>Write to cache line.<br>Update status to Dirty.                 | Evict dirty line to main memory.<br>Update line from owner, using read-for-ownership.<br>Write to cache line.<br>No status change.                                       |
| Write Hit              | Does not occur.  | Send invalidation signal to system bus.<br>Write to cache line.<br>Update status to Dirty.                                    | Write to cache line.<br>Update status to Dirty.  | Write to cache line.<br>No status change.  |
| <b>From System Bus</b> |  |   |  |  |
| Read Miss              | No response.<br>No status change.  | No response.<br>No status change.   | No response.<br>No status change.  | No response.<br>No status change.  |
| Read Hit               | Does not occur.  | Supply cache line to bus, if highest priority of caches snooped.<br>No status change.   | Supply cache line to bus.<br>Update status to Shared.  | Supply Dirty line to bus and write to main memory.<br>Update status to Shared.   |
| Read-for-Ownership Hit | Does not occur.  | Supply cache line to bus, if highest priority of caches snooped.<br>Update status to Invalid.                                 | Supply cache line to bus.<br>Update status to Invalid.   | Supply Dirty line to bus.<br>Update status to Invalid.   |
| Invalidate Hit         | Does not occur.  | Update status to Invalid.   | Does not occur.  | Does not occur.  |

Table 4.6. (continued)

|            |                                      |                                      |                                   |                                   |
|------------|--------------------------------------|--------------------------------------|-----------------------------------|-----------------------------------|
| Write Miss | No response.<br>No status<br>change. | No response.<br>No status<br>change. | No response.<br>No status change. | No response.<br>No status change. |
| Write Hit  | Does not occur.                      | Does not occur.                      | Does not occur.                   | Does not occur.                   |

The memory is given lowest priority to respond, and the highest priority device which receives a snoop hit will feed the data to the requesting cache even if the line is only in a Shared state. If that device has a Valid-Exclusive or Dirty copy, it downgrades that line's status to Shared. Also, if the snooped copy was Dirty, the responding cache intervenes directly, using a main memory write cycle to copy the line back into main memory simultaneously with the read cycle which caused the snoop hit. The initiating cache will see that the data did not come from main memory and will set the line's status to Shared. If no other caches respond, the main memory supplies the data to the requesting cache, which sees the signal indicating that the copy came from main memory, and sets the state of its line to Valid-Exclusive. As in Futurebus+, the notification of snoop hits can remove the need for a write-once cycle to be performed for certain lines, thus reducing the bus traffic.

A CPU read miss to a Shared or a Valid-Exclusive location is treated the same as the CPU read miss just described. If the missed line is Dirty, however, an eviction is instigated, but then is followed by the same CPU read miss cycle. Read hits, as in any other cache, cause no bus activity whatsoever, and the hit line's status is not changed. One slightly unusual possibility is that a Shared line can be overwritten without other processors noticing that one less copy of the cached line exists. This means that there will be times when a processor will have the only copy of a cache line and will have that line marked in a Shared state, even though it could rightfully have that copy of the line in a Valid-Exclusive state, which would allow it one faster write cycle should the processor ever want to write to that line. Although this might cause an extremely minor performance impact, it will not make the coherency protocol any less bulletproof.

Since the cache is implemented using a write allocation strategy, all CPU write miss cycles begin similarly to their equivalent read miss cycles, with the exception that a Read-for-Ownership cycle is used to indicate that the requesting cache intends to take the line immediately into a Dirty state. If the line is cached, the highest priority cache which contains a copy of this line will supply it to the requesting cache, and all caches which experience snoop hits will immediately invalidate their own copies. As opposed to most of the other coherency protocols reviewed here, the transition of a cache line from a Dirty state in one cache to a Dirty state in another cache is not a sign of

sick software. Write allocation schemes are typically used if the shortest CPU write cycle is less than the line length, so one cache/CPU could have only written to the least significant byte or word of a line, while the next cache/CPU would write to a more significant byte or word. Something else which is pretty unique about the Dirty-to-Dirty transition is that main memory does not need to be updated in this case. If reflection is not built into main memory, there is no reason to expect main memory to grasp a copy of the interim version of this line.

Only one type of CPU write hit cycle will produce any bus activity, and that is a CPU write hit to a Shared location. When a processor writes to a line which is already in the Shared state, the cache controller sends an invalidate cycle out on the main memory bus, causing all other Shared copies of the same cache line to immediately invalidate themselves. By the nature of the Illinois protocol, there can be no matching lines in any other states except the Shared state. The cache which sent the invalidation signal can now take its line into the Dirty state. A CPU write hit to a Valid-Exclusive or a Dirty location results in an immediate write cycle, and the line ends the cycle in the Dirty case.

Snoop write hits just plain don't happen since write allocation is used. Anything that is written is either in a noncacheable space or is written because a Dirty line was evicted, and a Dirty line will only be copied in a single cache, so no snoop hit can occur.

#### 4.3.7 Firefly

The DEC Firefly was an unusual implementation in that it never used an Invalid state. As Table 4.7 shows, there are only three states: Shared, Valid Exclusive, and Dirty. How does the cache work without an Invalid state? It's something like the cache we looked over in Section 2.1.3, which used no status bits whatsoever, but assured that all locations were valid after a cold start by disabling the cache until a start-up routine had given all cache lines a chance to be replaced. Unlike MESI and MOESI, write cycles from one CPU/cache don't cause invalidations of matching addresses in other caches. Instead, the Firefly protocol requires snooped caches to update their contents during snoop write hit cycles. All CPU write cycles to cached locations which are not in an Exclusive state are propagated to the main memory bus. Thus, CPU write cycles to cache lines which are not in an Exclusive state in the local cache update both main memory and all snooped cache locations. This is broadcasting and was described in Section 4.2.5.

This special protocol requires a bus signal to indicate that a snoop hit has occurred in another processor's cache (as do Futurebus+ and the University of Illinois protocol), and, as with Futurebus+, the University of Illinois pro-

Table 4.7. Firefly

|                        | Shared   | Valid Exclusive   | Dirty   |
|------------------------|--|---|---|
| <b>From CPU Bus</b>    |  |   |   |
| Read Miss              | Update line from owner.<br>If from another cache, no status change.<br>If from main memory, update status to Valid-Exclusive.  | Update line from owner.<br>If from another cache, update status to Shared.<br>If from main memory, no status change.  | Evict Dirty line to main memory.<br>Update line from owner.<br>If from another cache, update status to Shared.<br>If from main memory, update status to Valid Exclusive.  |
| Read Hit               | Read cache data.<br>No status change.  | Read cache data.<br>No status change.   | Read cache data.<br>No status change.   |
| Write Miss             | Update line from owner.<br>If from another cache, write-through to cache and main memory.<br>No status change.<br>If from main memory, write to cache only.<br>Update status to Dirty. | Update line from owner.<br>If from another cache, write-through to cache and main memory.<br>Update status to Shared.<br>If from main memory, write to cache only.<br>Update status to Dirty. | Evict Dirty line to main memory.<br>Update line from owner.<br>If from another cache, write-through to cache and main memory.<br>Update status to Shared.<br>If from main memory, write to cache only.<br>No status change. |
| Write Hit              | Write data to system bus.<br>Write to cache line.<br>If other caches respond to write, no status change.<br>If no other caches respond to write, update status to Valid-Exclusive.     | Write to cache line.<br>Update status to Dirty.   | Write to cache line.<br>No status change.   |
| <b>From System Bus</b> |  |   |   |
| Read Miss              | No response.<br>No status change.  | No response.<br>No status change.   | No response.<br>No status change.   |
| Read Hit               | Supply cache line to bus, indicating shared data.<br>No status change.   | Supply cache line to bus, indicating shared data.<br>Update status to Shared.   | Supply Dirty line to bus and write to main memory, indicating shared data.<br>Update status to Shared.  |
| Write Miss             | No response.<br>No status change.  | No response.<br>No status change.   | No response.<br>No status change.   |
| Write Hit              | Update contents of line.<br>Indicate shared data.  | Does not occur.   | Does not occur.   |

tol, Berkeley, and N+1, Firefly uses write allocation to reduce the complexity of having to deal with write snoop hits. All bus write cycles fall into one of three categories: 1) They are not in cacheable areas, so coherency is not an issue, 2) they are to a Shared cache line, so they are observed by the other caches which have a Shared copy of the same cache line, or 3) they have come from the eviction of a Dirty line, and, therefore, no other cache can own a copy. Any cache which observes a snoop hit will not only tell its own cache controller of this occurrence, but will signal the fact to the entire bus, similarly to the  $tf^*$  signal in the Futurebus+. This allows the CPU/cache which instigated the transaction to figure out how to handle the cache line. As opposed to most of the coherency protocols discussed in this chapter, only one type of read and write cycle is used, with the entire protocol being handled via the single snoop hit signal.

You by now have noticed that this cache, like many of the other protocols, behaves as if it were two different kinds of cache with two different kinds of write policies, depending on the current state of the line being written. Lines which are marked as Shared are treated as write-through, unless no other cache responds (with a snoop hit) to a main memory write cycle to one of these locations. Any line marked as Valid Exclusive or Dirty is treated as if it were in a single-processor copy-back cache until a read snoop hit is encountered. This ties neatly into the discussion in Section 4.2.3 about the ease of assuring coherency through a write-through strategy versus the improved use of bus bandwidth afforded by a copy-back cache. Only those cache lines which are truly Shared will use up precious bus bandwidth.

As with the Illinois protocol, an attempted main memory read cycle which is intervened because of a read snoop hit immediately causes the evicting cache to initiate a write command during the same memory cycle. This means that the main memory must respond to a combined read and write command as if it were simply a write cycle.

In summary, the cache behaves as if it were write-through for lines which are copied in other caches. If there is no other cached copy of a cached line, the cache uses a copy-back write strategy. All snoop hits are responded to by the snooped cache flagging the hit via and main memory bus. All snooped caches supply the line simultaneously. Bus crashes are not a problem since the snooped caches all respond during the same bus cycle.

Snoop hits to a Shared location do not change the status of the location. Snoop hits to Valid Exclusive or Dirty locations downgrade those locations to a Shared status. Lines can be upgraded to Valid Exclusive through an absence of a snoop hit during any shared read or write activity and are automatically loaded as Dirty during the read cycle of a write allocation. CPU write hit cycles to Valid Exclusive lines upgrade those lines' status to Dirty without bus interaction.

### 4.3.8 Dragon

Xerox PARC (Palo Alto Research Center, in California, the same people who brought us the graphic user interface of the Apple Macintosh) devised a multiprocessing protocol called the Dragon (Table 4.8). The protocol is another four-state version (Shared Clean, Shared Dirty, Valid Exclusive, and Dirty), yet it is similar to the Firefly and Futurebus+ in using a signal to indicate snoop hits on the bus, and, like the Firefly, it does not have an Invalid state. Unlike both the Firefly and the University of Illinois protocols, system bus read cycles don't suddenly convert to main memory write cycles. The only memory write cycles which appear on the system bus are noncacheable writes and evictions. Two states can result in evictions: Shared Dirty and Dirty. These are indeed ownership states, but the word "owned" appears to have been carefully avoided in the development of this terminology.

Table 4.8. Dragon

|                     | Shared Clean  | Shared Dirty   | Valid Exclusive   | Dirty   |
|---------------------|---|--|---|---|
| <b>From CPU Bus</b> |   |  |   |   |
| Read Miss           | Update line from owner.<br>If from another cache, no status change.<br>If from main memory, update status to Valid-Exclusive.   | Evict dirty line to main memory.<br>Update line from owner.<br>If from another cache, update status to Shared-Clean.<br>If from main memory, update status to Valid-Exclusive.           | Update line from owner.<br>If from another cache, update status to Shared-Clean.<br>If from main memory, no status change.  | Evict dirty line to main memory.<br>Update line from owner.<br>If from another cache, update status to Shared-Clean.<br>If from main memory, update status to Valid-Exclusive.  |
| Read Hit            | Read cache data.<br>No status change.   | Read cache data.<br>No status change.  | Read cache data.<br>No status change.   | Read cache data.<br>No status change.   |
| Write Miss          | Update line from owner.<br>If from another cache, write-through to local and remote caches.<br>Update status to Shared-Dirty.<br>If from main memory, write to cache only.<br>Update status to Dirty. | Update line from owner.<br>If from another cache, write-through to local and remote caches.<br>No status change.<br>If from main memory, write to cache only.<br>Update status to Dirty. | Update line from owner.<br>If from another cache, write-through to local and remote caches.<br>Update status to Shared-Dirty.<br>If from main memory, write to cache only.<br>Update status to Dirty. | Evict dirty line to main memory.<br>Update line from owner.<br>If from another cache, write-through to local and remote caches.<br>Update status to Shared-Dirty.<br>If from main memory, write to cache only.<br>No status change. |

Table 4.8. (continued)

|                        |   |  |   |  |
|------------------------|---|--|---|--|
| Write Hit              | Write data to other caches, but not main memory. Write to cache line. If other caches respond to write, update status to Shared-Dirty. If no other caches respond to write, update status to Dirty. | Write data to other caches, but not main memory. Write to cache line. If other caches respond to write, no status change. If no other caches respond to write, update status to Dirty. | Write to cache line. Update status to Dirty.                                      | Write to cache line. No status change.   |
| <b>From System Bus</b> |   |  |   |  |
| Read Miss              | No response. No status change.  | No response. No status change.   | No response. No status change.  | No response. No status change.   |
| Read Hit               | Indicate shared data on bus. Do not send bus data. No status change.  | Supply cache line to bus, indicating shared data. No status change.  | Indicate shared data on bus. Do not send bus data. Update status to Shared-Clean. | Supply Dirty line to bus, indicating shared data. Update status to Shared-Dirty. |
| Write Miss             | No response. No status change.  | No response. No status change.   | No response. No status change.  | No response. No status change.   |
| Cache Write Hit        | Update contents of line. Indicate shared data on bus.   | Update contents of line. Indicate shared data on bus.  | Does not occur.   | Does not occur.  |
| Memory                 | No response. No status change.  | Does not occur.  | Does not occur.   | Does not occur.  |

The bus does need to support a third kind of write cycle, one which updates other caches, but not main memory. By using this sort of write cycle (which is designed to be much faster than a main memory write cycle), the owner of a cache line can use a broadcast to update all other copies of the same cache line, as occurs in the Firefly. Like the Firefly and Futurebus+, snoop hits are signaled on the bus, and each line is set up to use either a write-through or a copy-back protocol during subsequent CPU write hit cycles, depending on the status of this signal during a bus transaction.

Going through the protocol step by step, cache read misses which do not generate snoop hits in other caches are loaded into the cache as Valid Exclusive lines. If they do generate snoop hits, the lines are loaded as Shared Clean (i.e., unowned). The responding caches take their lines from the Valid Exclusive state to the Shared Clean state, or from the Dirty state to the Shared Dirty state, but if the snooped lines were already Shared Clean or Shared Dirty, their status does not change. If the snooped line is in either

Dirty state, the read cycle is satisfied from the snooped cache. If the snooped line started out as either Valid Exclusive or Shared Clean, then the snooped cache does not place data on the bus, but leaves this task up to main memory. Naturally, the bus has been designed to allow the snooped cache to disallow a main memory response. Finally, if a CPU read miss is aimed at a Dirty or Shared Dirty line in the local cache, that line must be evicted. Read hits, quite naturally, generate no bus traffic and are satisfied from the local cache without incurring a state change.

On a CPU write miss, since a write allocation scheme is used, the same process is followed as was detailed for a read miss in the preceding paragraph, except that the line is loaded as Dirty if no other cache responds. If other caches respond, the requesting cache sets its line as Shared Dirty, and broadcasts a Cache Write cycle to the other caches. This is a special bus cycle which writes a line to other caches, but not to main memory. In response to this write cycle, any other Shared Dirty copy will change its status to Shared Clean. During the reading portion of this allocation, any Valid Exclusive snooped copy of this line would have changed states to Shared Clean already, and a Dirty snooped copy would have changed its status to Shared Dirty, so the Valid Exclusive and Dirty states will not be encountered during a Cache Write snoop hit.

If you look at this closely, you'll see that a CPU write miss to a location which is snooped as Dirty in another cache will first result in the Dirty copy being converted to Shared Dirty during the read portion of the allocation, then will be changed from Shared Dirty to Shared Clean during the Cache Write portion of the cycle. This is a pretty involved little two-step! In the case where the missed line in the requesting cache is also Dirty, it also amounts to a heavy overhead of bus traffic. A CPU write miss to a Dirty or Shared Dirty line which is replaced with a Dirty or Shared Dirty line read from another cache will require three bus cycles. First, an eviction must be performed to make room for the new line. Second, the requesting CPU/cache performs a read cycle, and the requested data is read into the requesting cache from the intervening cache. Last, the requesting cache performs a Cache Write to update the line in the intervening cache which satisfied the request in the first place!

Two kinds of write hit cycles emulate either a write-through or a copy-back protocol. If the line is already either Dirty or Valid Exclusive, the cache behaves as if it were a copy-back cache in a single-processor system and writes only to the cache, without causing any bus traffic. At the end of either cycle, the status of the line will be Dirty. If the line is Shared Clean or Shared Dirty, the write cycle is broadcast to all other caches, but not to main memory, via a Cache Write cycle. At the end of either of these cycles, the line's status will be Shared Dirty, unless no other cache responds, in which case,

the line can start to act as if it uses a copy-back strategy, and the status goes immediately to Dirty. Main memory is not updated until the Dirty line is evicted from the owning cache, the one which has a Dirty or Shared Dirty copy of the line. This is performed using a Memory Write cycle. Since the Memory Write cycle only occurs during an eviction or a write to a non-cacheable address, the reaction which it causes in a snooped cache is straightforward. The only possible status of such a cycle in another cache is Shared Clean, which only happens if the line being evicted is Shared Dirty. In this case, the eviction has no effect, since the evicted line already matches the contents of the same line in the other cache. The Shared Clean line need not be invalidated as it would in a bus which did not have as much coherency support. A Dirty line cannot have a counterpart in another cache, since Dirty is an Exclusive state.

#### 4.3.9 Others

There are several good design examples which you can research to see ways in which the problem of coherency can be addressed in copy-back multiprocessor systems. These include commercial microprocessors which are all well documented and use their own solutions for these exact problems, cache controllers, as well as the several minicomputer and mainframe architectures which can be researched in professional journals like the Association of Computing Machinery's *Transactions on Computer Systems* or even *Electronic Design Magazine*.

I have attempted to show enough alternatives here to get you thinking, but have not given any concrete numbers for one system's performance against another's for the same very good reasons I have shied away from statistics throughout this book. First, your software is different from any that has been used to run any cache statistics to date. This means that you would be leading yourself astray if you were to use another person's statistics. Second, your system is different from others, so the amount of gain in reducing main memory reads and writes will be important to your selection of read and write policies, line size, and even coherency protocol. By all means, do whatever you can to measure real statistics on a real system before committing to a cache design. Any other method would be a crap shoot.