

is no decision about which Way to replace a line in. Instructions are automatically put into the instruction cache, and data is automatically placed into the data cache. Fourth, the caches do not have to match each other in size. Instruction caches should be larger than data caches in most general computing applications, but when real number crunching happens in such applications as array processing, small instruction caches will be adequate, while large data caches are a requirement. Another unusual benefit is that altogether different policies can be used in the two caches. There is no reason why the data cache cannot be a highly associative design, while the instruction cache might be direct-mapped. Should you choose to implement a split-cache architecture, look through this book once to help decide on a set of policies for the instruction cache, then do it again for the data cache. Your design might look unorthodox, but will be more likely to outperform your competitors'.

You can look into different ways to split the cache. Most processors not only indicate whether the current access is for an instruction or data, but many will also disclose user/supervisor mode and whether or not the request is for a stack operation. You just might be able to realize a lot of the advantages of a multiway cache without incurring the speed headaches simply by using any or all of these status signals to enable different split caches.

2.2.4 Write-through vs. Copy-back

Way back in Section 2.1.3, I deferred explaining what action the cache took during a write cycle. This was because there are several means of handling write cycles in caches, or **write strategies**, and the decision impacts the cost as well as the complexity of the cache. Even in this section, I will be deferring until Chapter 4 the explanations of some of the more specific problems involving write cycles.

There are two fundamental strategies defining the behavior of the cache during a write cycle. One is called the **write-through**, or **store through (ST)** cache, and the other is a **copy-back**, **write back**, **deferred write**, **nonwrite-through**, or **store-in** cache (**SIC**). Given all these options, this book will exclusively use the terms write-through and copy-back, since they are both common and are not easily confused with one another. Naturally, once a write strategy is implemented in hardware, it becomes a **write policy**.

In a write-through cache design, either of two courses of action is taken. This is key in designs which can process some write cycles in no-wait states, some of which will be described in Section 2.2.6. In many designs, the cache is updated if a hit occurs, and the cache ignores the write cycle if a miss occurs. In others, the line is automatically invalidated (**write invalidate**). This

approach is used to get over certain speed limitations of the hardware. With a third alternative, a line in the cache is written to whether the write cycle is a hit or a miss. This last course of action is called a **write update** and is usually taken to remove the need to check for a cache hit before writing to a line in a direct-mapped design. The cache controller can begin the write cycle just as soon as the processor indicates that a write cycle has begun. The cache example of Figure 1.15 uses a write update policy. Naturally, in a multiway cache, updating cache lines on miss cycles would not work, since the controller would not know which Way to update until a cache hit or miss had been detected. Imagine the problem of having an updated and an old copy of the same address in the same line of two different Ways of the cache!

The difference in throughput between updating the cache on a miss versus not updating on a miss appears to not yet have been deeply explored. Intuitively, it seems that most programs would read data before writing it, with the exception of data being pushed onto the stack, and with the initialization of memory-based pointers, where an immediate value would be written to a main memory location before being read, then rewritten a number of times. If the cache designer has the option of choosing either of these two approaches, it would be wise to discuss it with the programmers. No matter which approach is used, the write-through cache *always* updates main memory during all write cycles.

The copy-back cache does *not* always update main memory, but reduces main memory accesses by writing data to the cache alone at a much higher speed than that data could be written to main memory. This has three major advantages. First, write cycles become much faster than they would be if a main memory cycle were the penalty for every CPU write; second, some write cycles, like loop counters and stack entries, will only be written to the main memory a fraction of the number of times that the CPU tries to write them. Third, the processor is on the main memory bus a lower percentage of the time. This last factor is a concern in tightly coupled multiprocessing systems, where the speed of the entire system depends on the main memory bus traffic (we'll handle this in depth in Section 4.2).

These advantages come at a cost, however. Housekeeping in copy-back caches requires a lot of thought, especially in multiprocessing systems. The most basic concern is how to handle the data which has been written to the cache, but not to the main memory. At some point, the main memory will need to be updated with the fresher data which is in the cache. An opportunity for this usually occurs when the updated line is to be removed from the cache. Obviously, if this data were simply overwritten, as it would be in a write-through cache, new data would be destroyed, and the data integrity of the entire program would suffer. Therefore, a method must be implemented for the updated line to be transferred to main memory once it is to be re-

SET VALID DIRTY			CACHE-TAG RAM	CACHE DATA RAM
ADDR	BIT	BIT		
0	1	1	Address Tag 0	Dirty Data
1	0	?	N/A	Garbage
2	1	0	Address Tag 2	Valid Data
3	1	0	Address Tag 3	Valid Data
4	1	1	Address Tag 4	Dirty Data
5	0	?	N/A	Garbage
6	1	1	Address Tag 6	Dirty Data
7	1	0	Address Tag 7	Valid Data

Figure 2.14. The Dirty bit signifies whether a cache line contains data which is more up to date than the matching address in the main memory.

moved from the cache. The process of writing data back to main memory when it is being replaced in the cache is called **eviction** or **deallocation**. (The evicted line is called the **victim**.) Some less-often-used terms for eviction cycles are copy-back, write-back, **copy-out**, and **victim write**. I'll stick with the word "eviction" so that there will be no misunderstanding about whether I'm talking about the cache's write policy or the cycle being processed.

One very simple method of implementing a copy-back cache would be to write every Valid line which was being replaced back into main memory, whether or not it had actually been written to by the processor. This would make the cache waste a considerable amount of bus bandwidth with unnecessary main memory write cycles, due to all the evictions of lines which had not been written to by the CPU. Another problem with this method is that all line replacements would take twice as long as would line replacements in a write-through cache, since write-through line replacements require only a main memory read cycle. To avoid this burden, the cache is usually implemented with a means to signify whether a line in cache is more current than the main memory location it represents. The simplest method is to use another bit for each line in the cache, and this bit is called the **Dirty bit**. Data which has been written in the cache, but has not been updated in the main memory is tagged as **Dirty** by the cache controller's setting this bit. Like the Valid bit, there is usually a Dirty bit for every line in the cache (Figure 2.14). During a cache miss cycle, the line to be replaced is examined, and, if its Dirty bit is set, the current contents of that cache line are evicted back into main memory.

The Dirty bit is not the only means of signifying the Dirty status of a cache line. More complex methods are in wide use and will be examined in Chapter 4. In general, these other methods take advantage of the fact that more

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

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

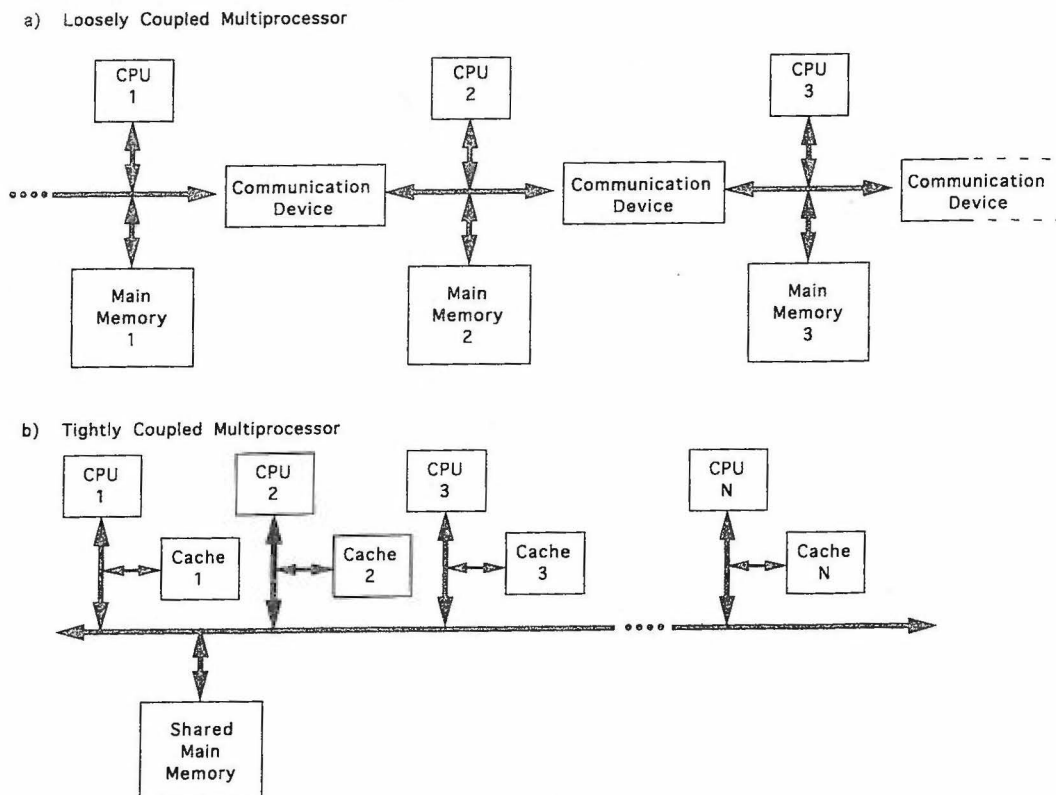


Figure 4.5. Examples of loosely coupled (a) and tightly coupled (b) multiprocessing systems. Main memory is shared in tightly coupled systems.

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.

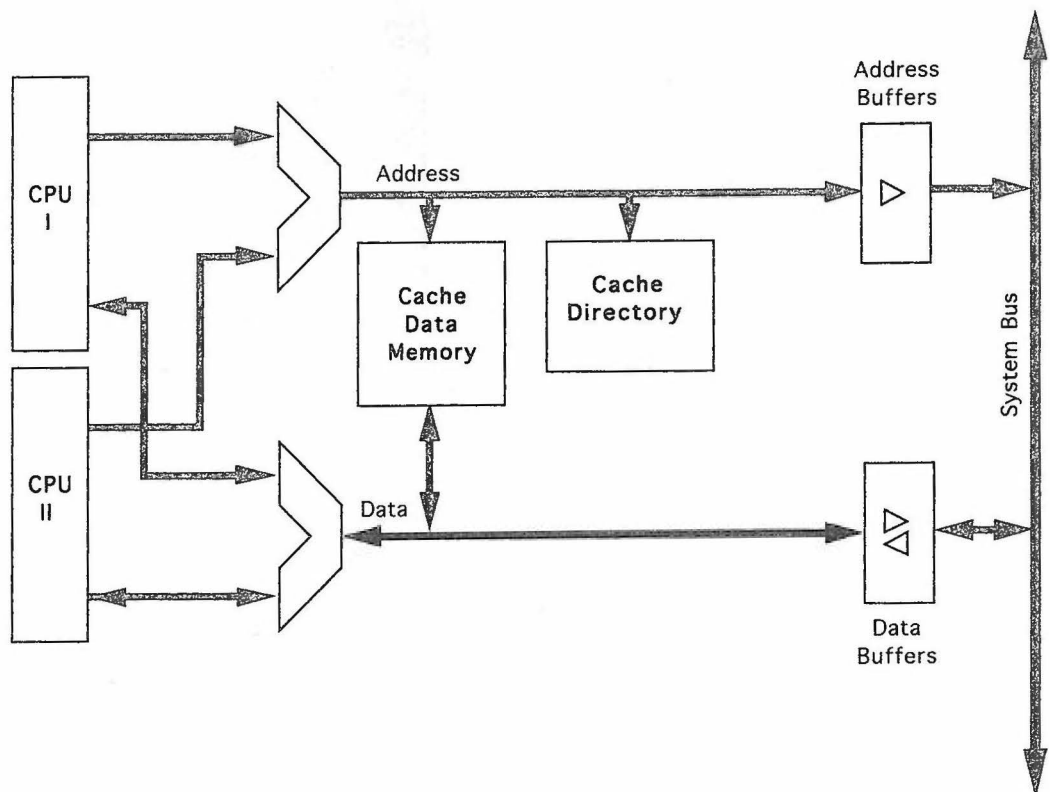


Figure 4.6. One cache serving two CPUs. This tends to cause thrashing since the two CPUs are in contention over the allocation of the cache.

cy-
ne
ys-
le
te.
le
ac
st-
an
Js
le-
ed
is
ne
c-
in
ts
e,



Js

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	Valid	Dirty	Valid	Dirty	Valid	Dirty		
1	0	1	0	0	0	0	0	0	0	1	0	Line 0	Valid copies in A, B, and F.
0	0	0	0	0	0	1	1	0	0	0	0	Line 1	D has only valid copy, and has written to it.
0	0	0	0	0	0	1	1	1	0	1	0	Line 2	D, E, and F have copies, and D has written to all.
0	0	0	0	0	0	0	0	0	0	0	0	Line 3	No cached copies exist.
0	0	1	0	0	0	0	0	0	0	0	0	Line 4	Cache B has only copy.
1	0	0	0	0	0	0	0	0	0	1	0	Line 5	Valid copies in A and F only.
0	0	1	0	1	0	1	0	1	0	1	0	Line 6	All except A have valid copies.
1	0	1	1	1	0	1	0	1	0	1	0	Line 7	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

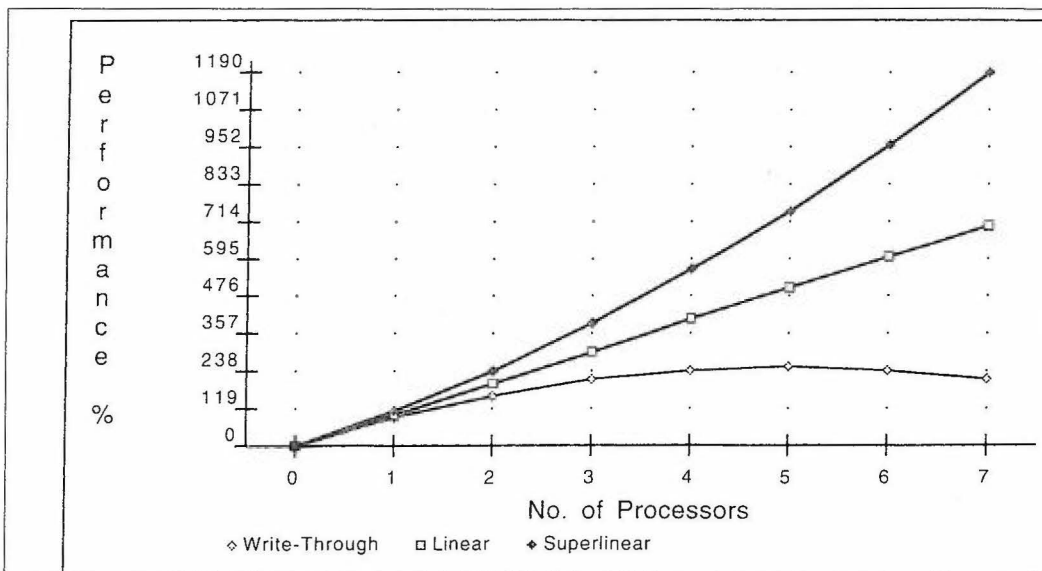


Figure 4.9. Throughput versus number of processors for three kinds of multiprocessing systems. The lower curve shows the effects of main memory bandwidth limitations, the middle line is an idealized linear ratio of performance to number of CPUs, and the top line shows superlinear performance, which can 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
From CPU Bus				
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.
From System Bus				
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.