Processor Action	State in P_1	State in P_2	State in P_3	Bus Action	Data Supplied By
1. P_1 reads u	E	—	—	BusRd	Memory
2. P_3 reads u	Sc	—	Sc	BusRd	Memory
3. P_3 writes u	Sc	—	Sm	BusUpd	P_3 cache
4. P_1 reads u	Sc	—	Sm	null	—
5. P_2 reads u	Sc	Sc	Sm	BusRd	P_3 cache

FIGURE 5.17 The Dragon update protocol in action for the processor actions shown in Figure 5.3. The figure shows the state of the relevant memory block at the end of each processor action, the bus transaction generated (if any), and the entity supplying the data.

atomic bus, a lot like they were in the write-through case. However, with both invalidation- and update-based protocols, we must address many subtle implementation issues and race conditions, even with an atomic bus and a single-level cache. We discuss this next level of protocol and hardware design in Chapter 6, as well as more realistic scenarios with pipelined buses, multilevel cache hierarchies, and hardware techniques that can reorder the completion of memory operations. Nonetheless, we can quantify many protocol trade-offs even at the state diagram level that we have been considering so far.

5.4 ASSESSING PROTOCOL DESIGN TRADE-OFFS

Like any other complex system, the design of a multiprocessor requires many interrelated decisions to be made. Even when a processor has been picked, we must decide on the maximum number of processors to be supported by the system, various parameters of the cache hierarchy (e.g., number of levels in the hierarchy, and for each level the cache size, associativity, block size, and whether the cache is write through or write back), the design of the bus (e.g., width of the data and address buses, the bus protocol), the design of the memory system (e.g., interleaved memory banks or not, width of memory banks, size of internal buffers), and the design of the I/O subsystem. Many of the issues are similar to those in uniprocessors (Smith 1982) but accentuated. For example, a write-through cache standing before the bus may be a poor choice for multiprocessors because the bus bandwidth is shared by many processors, and memory may need to be more greatly interleaved because it services cache misses from multiple processors. Greater cache associativity may also be useful in reducing conflict misses that generate bus traffic.

The cache coherence protocol is a crucial new design issue for a multiprocessor. It includes protocol class (invalidation or update), protocol states and actions, and lower-level implementation trade-offs. Protocol decisions interact with all the other design issues. On the one hand, the protocol influences the extent to which the latency and bandwidth characteristics of system components are stressed; on the other, the performance characteristics as well as the organization of the memory and communication architecture influence the choice of protocols. As discussed in

Chapter 4, these design decisions need to be evaluated relative to the behavior of real programs. Such evaluation was very common in the late 1980s, albeit using an immature set of parallel programs as workloads (Archibald and Baer 1986; Agarwal and Gupta 1988; Eggers and Katz 1988, 1989a, 1989b).

Making design decisions in real systems is part art and part science. The art draws on the past experience, intuition, and aesthetics of the designers, and the science is based in workload-driven evaluation. The goals are usually to meet a cost-performance target and to have a balanced system, so that no individual resource is a performance bottleneck yet each resource has only minimal excess capacity. This section illustrates some key protocol trade-offs by putting the workload-driven evaluation methodology from Chapter 4 into action.

5.4.1 Methodology

The basic strategy is as follows. The workload is executed on a simulator of a multiprocessor architecture, as described in Chapter 4. By observing the state transitions encountered in the simulator, we can determine the frequency of various events such as cache misses and bus transactions. We can then evaluate the effect of protocol choices in terms of other design parameters such as latency and bandwidth requirements.

Choosing parameters according to the methodology of Chapter 4, this section first establishes the basic state transition characteristics generated by the set of applications for the four-state Illinois MESI protocol. It then illustrates how to use these frequency measurements to obtain a preliminary quantitative analysis of the design trade-offs raised by the example protocols above, such as the use of the exclusive state in the MESI protocol and the use of BusUpgr rather than BusRdX transactions for the S $\rightarrow$ M transition. This section also illustrates more traditional design issues, such as how the cache block size—the granularity of both coherence and communication—impacts the latency and bandwidth needs of the applications. To understand this effect, we classify cache misses into categories such as cold, capacity, and sharing misses, examine the effect of block size on each category, and explain the results in light of application characteristics. Finally, this understanding of the applications is used to illustrate the trade-offs between invalidation-based and update-based protocols, again in light of latency and bandwidth implications.

The analysis in this section is based on the frequency of various important events, not on the absolute times taken or, therefore, the performance. This approach is common in studies of cache architecture because the results transcend particular system implementations and technology assumptions. However, it should be viewed as only a preliminary analysis since many detailed factors that might affect the performance trade-offs in real systems are abstracted away. For example, measuring state transitions provides a means of calculating miss rates and bus traffic, but realistic values for latency, overhead, and occupancy are needed to translate the rates into the actual bandwidth requirements imposed on the system. To obtain an estimate of bandwidth requirements, we may artificially assume that every reference takes a fixed number of cycles to complete. However, the bandwidth requirements themselves do not translate into performance directly but only indirectly by increasing the cost of misses due to contention. Contention is very difficult to estimate because it depends on the timing parameters used and on the burstiness of the traffic, which is not captured by the frequency measurements. Contention, timing, and hence performance are also affected by lower-level interactions with hardware structures (like queues and buffers) and policies.

The simulations used in this section do not model contention. Instead, they use a simple PRAM cost model: all memory operations are assumed to complete in the same amount of time (here a single cycle) regardless of whether they hit or miss in the cache. There are three main reasons for this. First, the focus is on understanding inherent protocol behavior and trade-offs in terms of event frequencies, not so much on performance. Second, since we are experimenting with different cache block sizes and organizations, we would like the interleaving of references from application processes on the simulator to be the same regardless of these choices; that is, all protocols and block sizes should see the same trace of references. With the execution-driven rather than trace-driven simulation we use, this is only possible if we make the cost of every memory operation the same in the simulations. Otherwise, if a reference misses with a small cache block but hits with a larger one, for example, then it will be delayed by different amounts in the interleaving in the two cases. It would therefore be difficult to determine which effects are inherently due to the protocol and which are due to the particular parameter values chosen. Third, realistic simulations that model contention take much more time. The disadvantage of using this simple model even to measure frequencies is that the timing model may affect some of the frequencies we observe; however, this effect is small for the applications we study.

The illustrative workloads we use are the six parallel programs (from the SPLASH-2 suite) and one multiprogrammed workload described in Chapters 3 and 4. The parallel programs run in batch mode with exclusive access to the machine and do not include operating system activity in the simulations, whereas the multiprogrammed workload includes operating system activity. The number of applications used is relatively small, but the applications are primarily for illustration as discussed in Chapter 4; the emphasis here is on choosing programs that represent important classes of computation and with widely varying characteristics. The frequencies of basic operations for the applications appear in Table 4.1. We now study them in more detail to assess design trade-offs in cache coherency protocols.

5.4.2 Bandwidth Requirement under the MESI Protocol

We begin by using the default 1-MB, single-level caches per processor, as discussed in Chapter 4. These are large enough to hold the important working sets for the default problem sizes, which is a realistic scenario for all applications. We use four-way set associativity (with LRU replacement) to reduce conflict misses and a 64-byte cache block size for realism. Driving the workloads through a cache simulator that models the Illinois MESI protocol generates the state transition frequencies shown in Table 5.1. The data is presented as the number of state transitions of a particular type per 1,000 references issued by the processors. Note in the table that a new state,

Table 5.1 State Transitions per 1,000 Data Memory References Issued by the Applications

Application	From	To: NP	I	E	S	M
Barnes-Hut	NP	0	0	0.0011	0.0362	0.0035
	I	0.0201	0	0.0001	0.1856	0.0010
	E	0.0000	0.0000	0.0153	0.0002	0.0010
	S	0.0029	0.2130	0	97.1712	0.1253
	M	0.0013	0.0010	0	0.1277	902.782
LU	NP	0	0	0.0000	0.6593	0.0011
	I	0.0000	0	0	0.0002	0.0003
	E	0.0000	0	0.4454	0.0004	0.2164
	S	0.0339	0.0001	0	302.702	0.0000
	M	0.0001	0.0007	0	0.2164	697.129
Ocean	NP	0	0	1.2484	0.9565	1.6787
	I	0.6362	0	0	1.8676	0.0015
	E	0.2040	0	14.0040	0.0240	0.9955
	S	0.4175	2.4994	0	134.716	2.2392
	M	2.6259	0.0015	0	2.2996	843.565
Radiosity	NP	0	0	0.0068	0.2581	0.0354
	I	0.0262	0	0	0.5766	0.0324
	E	0	0.0003	0.0241	0.0001	0.0060
	S	0.0092	0.7264	0	162.569	0.2768
	M	0.0219	0.0305	0	0.3125	839.507
Radix	NP	0	0	0.004746	3.524705	11.41111
	I	0.130988	0	0	1.108079	4.57868
	E	0.000759	0.002848	0.080301	0	0.00019
	S	0.029804	1.120988	0	178.1932	0.817818
	M	0.044232	11.53127	0	4.03157	802.282

continued

Table 5.1 State Transitions per 1,000 Data Memory References Issued by the Applications

		To				
Application		NP	I	E	S	M
Raytrace	From NP	0	0	1.3358	1.5486	0.0026
	From I	0.0242	0	0.0000	0.3403	0.0000
	From E	0.8663	0	29.0187	0.3639	0.0175
	From S	1.1181	0.3740	0	310.949	0.2898
	From M	0.0559	0.0001	0	0.2970	661.011
Multiprog User Data References	From NP	0	0	0.1675	0.5253	0.1843
	From I	0.2619	0	0.0007	0.0072	0.0013
	From E	0.0729	0.0008	11.6629	0.0221	0.0680
	From S	0.3062	0.2787	0	214.6523	0.2570
	From M	0.2134	0.1196	0	0.3732	772.7819
Multiprog User Instruction References	From NP	0	0	3.2709	15.7722	0
	From I	0	0	0	0	0
	From E	1.3029	0	46.7898	1.8961	0
	From S	16.9032	0	0	981.2618	0
	From M	0	0	0	0	0
Multiprog Kernel Data References	From NP	0	0	1.0241	1.7209	4.0793
	From I	1.3950	0	0.0079	1.1495	0.1153
	From E	0.5511	0.0063	55.7680	0.0999	0.3352
	From S	1.2740	2.0514	0	393.5066	1.7800
	From M	3.1827	0.3551	0	2.0732	542.4318
Multiprog Kernel Instruction References	From NP	0	0	2.1799	26.5124	0
	From I	0	0	0	0	0
	From E	0.8829	0	5.2156	1.2223	0
	From S	24.6963	0	0	1,075.2158	0
	From M	0	0	0	0	0

The data assumes 16 processors (except for Multiprog, which is for 8 processors), 1-MB four-way set-associative caches, 64-byte cache blocks, and the Illinois MESI coherence protocol.

NP (not present), is introduced. This addition helps clarify transitions where, on a cache miss, one block is replaced (creating a transition from one of I, E, S, or M to NP) and a new block is brought in (creating a transition from NP to one of I, E, S, or M). The sum of state transitions can be greater than 1,000 even though we are presenting averages per 1,000 references because some references cause multiple state transitions. For example, a write miss can cause two transitions in the local processor's cache (e.g., S → NP for the old block and NP → M for the incoming block), in addition to transitions in other caches due to invalidations (I/E/S/M → I). This state transition frequency data is very useful for answering "what if" questions. Example 5.8 shows how we can determine the bandwidth requirement these workloads would place on the memory system.

EXAMPLE 5.8 Suppose that the integer-intensive applications run at a sustained 200 MIPS per processor and the floating-point-intensive applications at 200 MFLOPS per processor. Assuming that cache block transfers move 64 bytes on the data bus lines and that each bus transaction involves 6 bytes of command and address on the address lines, what is the traffic generated per processor?

Answer The first step is to calculate the amount of traffic per instruction. We determine what bus action is taken for each of the possible state transitions and therefore how much traffic is associated with each transaction. For example, an M → NP transition indicates that, due to a miss, a modified cache block needs to be written back. Similarly, an S → M transition indicates that an upgrade request must be issued on the bus. Flushing a modified block response to a bus transaction (e.g., the M → S or M → I transition) leads to a BusWB transaction as well. The bus transactions for all possible transitions are shown in Table 5.2. All transactions generate 6 bytes of address bus traffic and 64 bytes of data traffic, except BusUpgr, which only generates address traffic. We can now compute the traffic generated. Using Table 5.2, we can convert the state transitions per 1,000 memory references in Table 5.1 to bus transactions per 1,000 memory references and convert this to address and data traffic by multiplying by the traffic per transaction. Then, using the frequency of memory accesses in Table 4.1, we can convert this to traffic per instruction. Finally, multiplying by the assumed processing rate, we get the address and data bandwidth requirement for each application. The result of this calculation is shown by the leftmost bar for each application in Figure 5.18.[5] ■

5. For the Multiprog workload, to speed up the simulations, a 32-KB instruction cache is used as a filter before passing the instruction references to the 1-MB unified instruction and data cache. The state transition frequencies for the instruction references are computed based only on those references that missed in the L_1 instruction cache. This filtering does not affect how we compute data traffic, but it means that instruction traffic is computed differently. In addition, for Multiprog we present data separately for kernel instructions, kernel data references, user instructions, and user data references. A given reference may produce transitions of multiple types for user and kernel data. For example, if a kernel instruction miss causes a modified user data block to be written back, then we will have one transition for kernel instructions from NP → E/S and another transition for the user data reference category from M → NP.

Table 5.2 Bus Actions Corresponding to State Transitions in Illinois MESI Protocol

		To				
		NP	I	E	S	M
From	NP	—	—	BusRd	BusRd	BusRdX
	I	—	—	BusRd	BusRd	BusRdX
	E	—	—	—	—	—
	S	—	—	Not possible	—	BusUpgr
	M	BusWB	BusWB	Not possible	BusWB	—

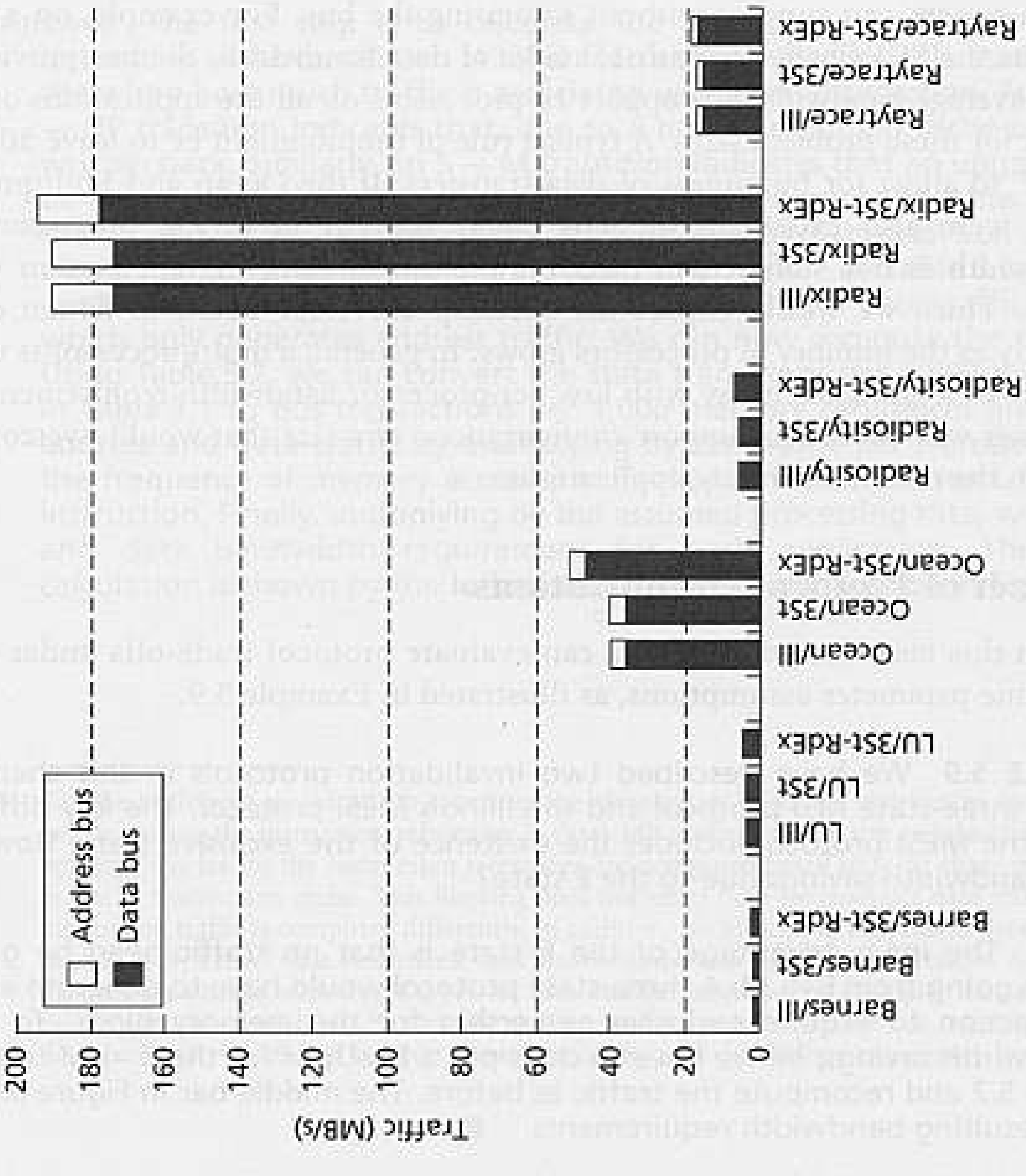
The calculation in the preceding example gives the average bandwidth requirement under the assumption that the bus bandwidth is enough to allow the processors to execute at full speed. (In practice, bandwidth limitations may slow processors and events down, which in turn would lead to lower traffic per unit time.) This calculation provides a useful basis for sizing the number of processors that a system can support without saturating the bus. For example, on a machine such as the SGI Challenge with 1.2 GB/s of data bandwidth, the bus provides sufficient average bandwidth to support 16 processors on all the applications other than Radix for these problem sizes. A typical rule of thumb might be to leave 50% "headroom" to allow for burstiness of data transfers. If the Ocean and Multiprog workloads were also excluded, the bus could support up to 32 processors. If the bandwidth is not sufficient to support the application, the application will slow down. Thus, we would expect the speedup curve for Radix to flatten out quite quickly as the number of processors grows. In general, a multiprocessor is used for a variety of workloads, many with low per-processor bandwidth requirements, so the designer will choose to support configurations of a size that would overcommit the bus on the most demanding applications.

5.4.3 Impact of Protocol Optimizations

Given this base design point, we can evaluate protocol trade-offs under common machine parameter assumptions, as illustrated in Example 5.9.

EXAMPLE 5.9 We have described two invalidation protocols in this chapter—the basic three-state MSI protocol and the Illinois MESI protocol. The key difference is that the MESI protocol includes the existence of the exclusive state. How large is the bandwidth savings due to the E state?

Answer The main advantage of the E state is that no traffic need be generated when going from E → M. A three-state protocol would have to generate a BusUpgr transaction to acquire exclusive ownership for the memory block. To compute bandwidth savings, all we have to do is put a BusUpgr for the E → M transition in Table 5.2 and recompute the traffic as before. The middle bar in Figure 5.18 shows the resulting bandwidth requirements. ■



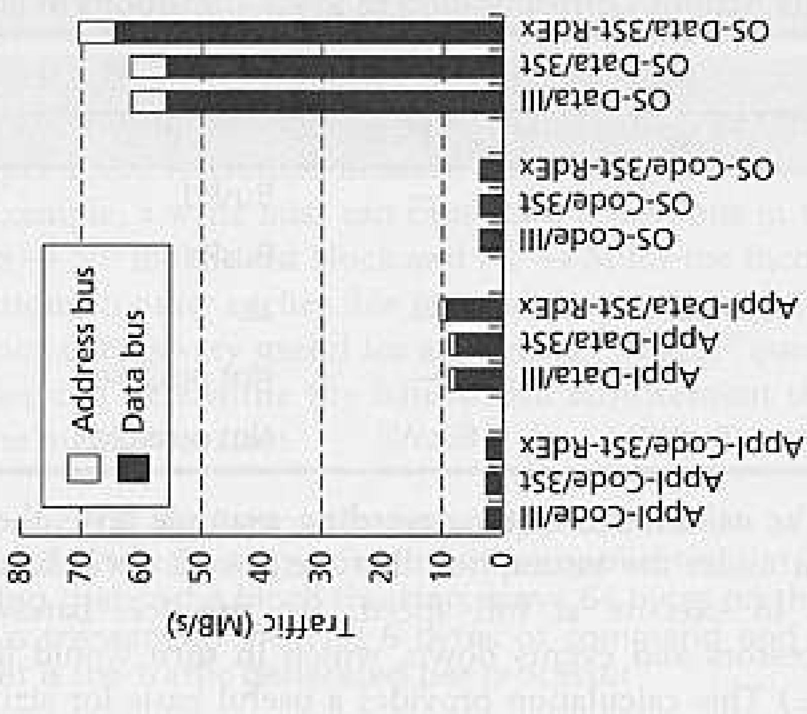


FIGURE 5.18 Per-processor bandwidth requirements for the various applications, assuming 200-MIPS/MFLOPS processors and 1-MB caches per processor. The left bar chart shows data for the parallel programs, and the right chart shows data for the Multiprog workload. The traffic is split into data traffic and address (including command) bus traffic. The leftmost bar shows traffic for the Illinois MESI protocol (Ill), the middle bar for the case where we use the basic three-state invalidation protocol without the E state (3St), and the rightmost bar for the three-state protocol when we use BusRdX instead of BusUpgr for S $\rightarrow$ M transitions (3St-RdEx).

Example 5.9 illustrates how an intuitive rationale for a more complex design may not stand up to quantitative measurement of workloads. Contrary to expectations, the E state offers negligible savings in traffic. This is true even for the Multiprog workload, which consists primarily of sequential jobs and should have benefited most. The primary reason for this negligible gain is that the fraction of E $\rightarrow$ M transitions in Table 5.1 is quite small (i.e., blocks loaded in exclusive state by a read miss are not often written while still in that state). In addition, the BusUpgr transaction that would have been needed for the S $\rightarrow$ M transition in a three-state protocol takes only 6 bytes of address traffic and no data traffic. Example 5.10 examines the advantage of the BusUpgr transaction.

EXAMPLE 5.10 Recall that even in the three-state MSI protocol, a write that finds the memory block in shared state in the cache generates a BusUpgr request on the bus rather than a BusRdX. This saves bandwidth, as no data need be transferred for a BusUpgr, but it complicates the implementation, as we shall see. The question is, how much bandwidth are we saving for taking on the extra complexity?

Answer To compute the bandwidth for the less complex implementation and a three-state protocol, all we have to do is put in BusRdX in the E $\rightarrow$ M and S $\rightarrow$ M transitions in Table 5.2 (these would all be S $\rightarrow$ M transitions in the three-state MSI protocol) and then recompute the bandwidth numbers. The results for all applications are shown in the rightmost bar in Figure 5.18. While for most applications the difference in bandwidth is small, Ocean and Multiprog kernel data references show that it can be as large as 10–20% for some applications. ■

The performance impact of these differences in bandwidth requirement depends on how the bus transactions are actually implemented. However, this high-level analysis indicates where more detailed evaluation is required.

Finally, as discussed in Chapter 4, for the input data set sizes we are using it is important that we run the Ocean, Raytrace, and Radix applications for smaller cache sizes as well, to model the situation where an important working set does not fit in the cache hierarchy. We use 64-KB caches here, which fit all but the largest working set for these problem sizes. The raw state transition data for this case is presented in Table 5.3, and the per-processor bandwidth requirements are shown in Figure 5.19. As we can see, not having one of the critical working sets fit in the processor cache can dramatically increase the bus bandwidth required due to capacity misses. A 1.2-GB/s bus can now barely support 4 processors for Ocean and Radix and 16 processors for Raytrace.

5.4.4 Trade-Offs in Cache Block Size

The cache organization is a critical performance factor in all modern computers, but it is especially so in multiprocessors. In the uniprocessor context, cache misses are typically categorized into the "three Cs": compulsory, capacity, and conflict misses (Hill and Smith 1989; Hennessy and Patterson 1996). *Compulsory misses*, or *cold misses*, occur on the first reference to a memory block by a processor. *Capacity misses* occur when all the blocks that are referenced by a processor during the execution of a program do not fit in the cache (even with full associativity), so some

Table 5.3 State Transitions per 1,000 Memory References Issued by the Applications with Smaller Caches

		To				
Application	From	NP	I	E	S	M
Ocean	NP	0	0	26.2491	2.6030	15.1459
	I	1.3305	0	0	0.3012	0.0008
	E	21.1804	0.2976	452.580	0.4489	4.3216
	S	2.4632	1.3333	0	113.257	1.1112
	M	19.0240	0.0015	0	1.5543	387.780
Radix	NP	0	0	9.440787	2.557865	27.36084
	I	4.354862	0	0.00057	0.157565	1.499903
	E	8.148377	0.001329	140.9295	0.012339	0.126621
	S	3.825407	0.481427	0	102.4144	0.484464
	M	23.03084	5.629429	0	2.069604	717.1426
Raytrace	NP	0	0	7.2642	3.9742	0.1305
	I	0.0526	0	0.0003	0.2799	0.0000
	E	6.4119	0	131.944	0.7973	0.0496
	S	4.6768	0.3329	0	205.994	0.2835
	M	0.1812	0.0001	0	0.2837	660.753

The data assumes 16 processors, 64-KB four-way set-associative caches, 64-byte cache blocks, and the Illinois MESI coherence protocol.

blocks are replaced and later accessed again. *Conflict* or *collision misses* occur in caches with less than full associativity when the collection of blocks referenced by a program that maps to a single cache set does not fit in the set. They are misses that would not have occurred in a fully associative cache. Many studies have examined how cache size, associativity, and block size affect each category of miss.

Architecturally, capacity misses are reduced by enlarging the cache. Conflict misses are reduced by increasing the associativity or increasing the number of lines to map to in the cache (by increasing cache size or reducing block size). Cold misses can be reduced only by increasing the block size so that a single cold miss will bring in more data that may be accessed thereafter as well. What makes cache design challenging in uniprocessors is that these factors trade off against one another. For example, increasing the block size for a fixed cache capacity will reduce the number of blocks, so the reduced cold misses may come at the cost of increased conflict misses. Also, variations in cache organization can affect the miss penalty or the hit time and, therefore, perhaps the processor cycle time.

Cache-coherent multiprocessors introduce a fourth category of misses: *coherence misses*. These occur when blocks of data are shared among multiple caches. There

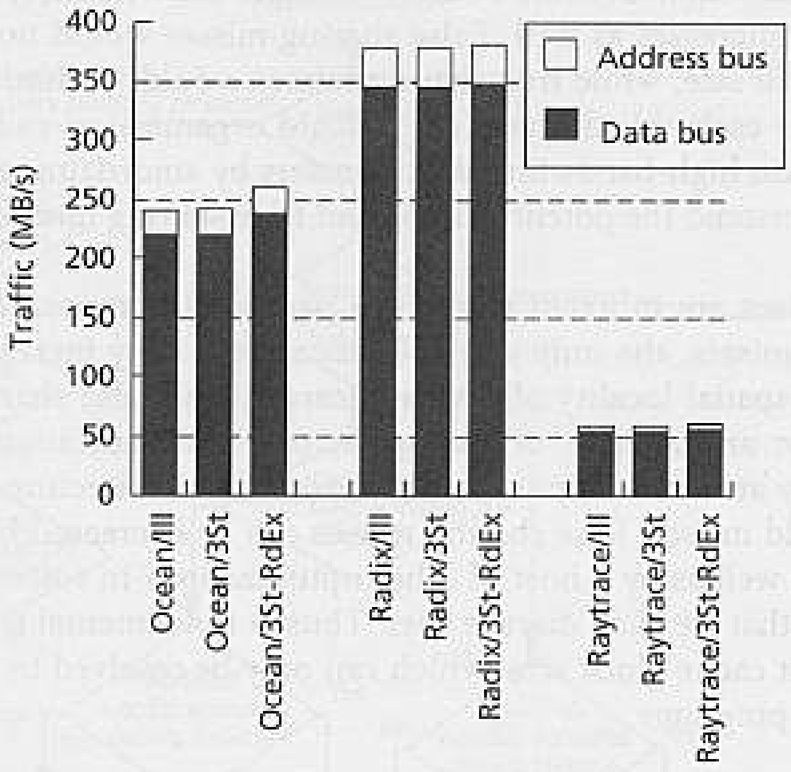


FIGURE 5.19 Per-processor bandwidth requirements for the various applications, assuming 200-MIPS/MFLOPS processors and 64-KB caches. The traffic is split into data traffic and address (including command) bus traffic. The leftmost bar shows traffic for the Illinois MESI protocol, the middle bar for the case where we use the basic three-state invalidation protocol without the E state (as described in Section 5.3.1), and the rightmost bar for the three-state protocol when we use BusRdX instead of BusUpgr for S → M transitions.

are two types: true sharing and false sharing misses. True sharing occurs when a data word produced (written) by one processor is used (read or written) by another. False sharing occurs when independent data words accessed by different processors happen to be placed in the same memory (cache) block, and at least one of the accesses is a write. The cache block size is not only the granularity (or unit) of the data fetched from the main memory, it is also typically used as the granularity of coherence. That is, on a write by a processor, the whole cache block is invalidated in other processors' caches, not just the word that is written.

More precisely, a *true sharing miss* occurs when one processor writes some words in a cache block, invalidating that block in another processor's cache, after which the second processor reads one of the modified words. It is called a "true" sharing miss because the miss truly communicates newly defined data values that are used by the second processor; such misses are essential to the correctness of the program, regardless of interactions with the machine organization or granularities. On the other hand, when one processor writes a word in a cache block and then another processor reads (or writes) a different word in the same cache block, the invalidation of the block and subsequent cache miss occurs as well, even though no useful values are being communicated between the processors. These misses are thus called *false sharing misses* (Dubois et al. 1993). As cache block size is increased, the probability of distinct variables being accessed by different processors but residing on the same cache block increases. If at least some of these variables are written, the likelihood of false sharing misses increases as well. False sharing misses would not occur with a one-word cache block size, while true sharing misses would. Technology pushes in the direction of large cache block sizes (e.g., DRAM organization and access modes and the need to obtain high-bandwidth data transfers by amortizing overhead), so it is important to understand the potential impact of false sharing misses and how they may be avoided.

True sharing misses are inherent to a given parallel decomposition and assignment, so, like cold misses, the only way to reduce them is by increasing the block size and increasing spatial locality of communicated data. False sharing misses, on the other hand, are an example of the artifactual communication discussed in Chapter 3 since they are caused by interactions with the architecture. In contrast to true sharing and cold misses, false sharing misses can be decreased by reducing the cache block size, as well as by a host of other optimizations in software (orchestration) and hardware that we shall discuss later. Thus, a fundamental tension exists in determining the best cache block size, which can only be resolved by evaluating the options against real programs.

A Classification of Cache Misses

The flowchart in Figure 5.20 gives a detailed algorithm for classifying cache misses in cache-coherent multiprocessors.[6] Understanding the details is not critical for now—it is enough for the rest of the chapter to understand only the preceding definitions—but it adds insight and is a useful exercise. In the algorithm, the *lifetime* of a block in a cache is defined as the time interval during which the block remains valid in the cache, that is, the time from the occurrence of the miss that loads the block in the cache until its invalidation, replacement, or the end of the program. We cannot classify a cache miss when it occurs but only when the fetched memory block is replaced or invalidated in the cache, because it is only then that we know whether true sharing or only false sharing occurred during that lifetime. Let us consider the simple cases first. Cases 1 and 2 are straightforward cold misses occurring on previously unwritten blocks. Cases 7 and 8 reflect false and true sharing on a block that was previously invalidated in the cache but yet replaced by another. The type of sharing is determined by whether the specific word or words modified since the invalidation are actually used during the current lifetime. Case 9 is a straightforward capacity (or conflict) miss since the block was previously replaced from the cache and the words in the block have not been modified since last accessed. All of the other cases refer to misses that occur due to a combination of factors. For example, cases 4 and 5 are cold misses because this processor has never accessed the block before; however, some other processor had written the block, so there is also

6. In this classification, we do not distinguish conflict from capacity misses since both are a result of the available resources (set or entire cache) becoming full and the difference between them does not shed additional light on multiprocessor issues.

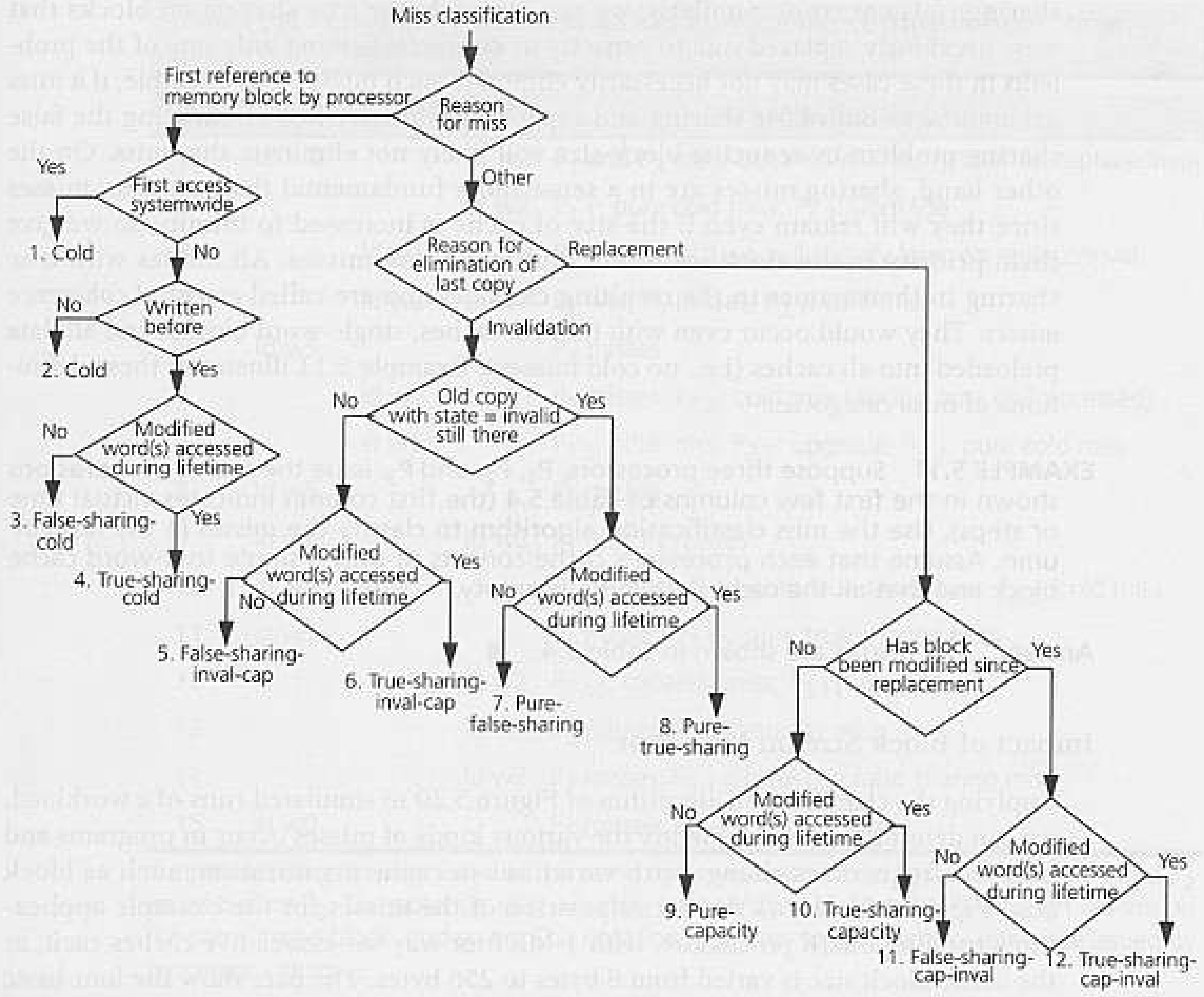

Miss classification
Reason for miss
First reference to memory block by processor
Other
First access systemwide
Yes
No
1. Cold
Written before
No
Yes
2. Cold
Modified word(s) accessed during lifetime
No
Yes
3. False-sharing-cold
4. True-sharing-cold
Reason for elimination of last copy
Replacement
Invalidation
Old copy with state = invalid still there
No
Yes
Modified word(s) accessed during lifetime
No
Yes
5. False-sharing-inval-cap
6. True-sharing-inval-cap
Modified word(s) accessed during lifetime
No
Yes
7. Pure-false-sharing
8. Pure-true-sharing
Has block been modified since replacement
No
Yes
Modified word(s) accessed during lifetime
No
Yes
9. Pure-capacity
10. True-sharing-capacity
Modified word(s) accessed during lifetime
No
Yes
11. False-sharing-cap-inval
12. True-sharing-cap-inval

FIGURE 5.20 A classification of cache misses for shared memory multiprocessors. The four basic categories of cache misses in this classification are cold, capacity, true sharing, and false sharing misses (conflict misses are considered to be capacity misses for this purpose). Many mixed categories arise because there may be multiple causes for a miss. For example, a block may be first replaced from processor *A*'s cache, then written to by processor *B*, and then read back by processor *A*, making it a capacity-cum-invalidation false/true sharing miss. This would be labeled "false/true sharing cap-inval" in the classification since sharing takes priority and since the replacement happened before the invalidation (cases 11 and 12 in the figure). If the block were first invalidated in *A*'s cache, then the invalid block replaced, and then read again by *A*, it would be labeled "false/true sharing inval-cap" (cases 6 and 7). In terms of the four major categories, these misses all fall into true or false sharing misses, as appropriate. *Note*: the question "modified word(s) accessed during lifetime?" asks whether accesses are made by this processor in the current lifetime to word(s) within the cache block that have been modified since the last "essential coherence" miss to this block by this processor, where essential coherence misses correspond to categories 4, 6, 8, 10, and 12. This can only be determined when the current lifetime of the block ends.

sharing (false or true). Similarly, we can have false or true sharing on blocks that were previously replaced due to capacity or conflicts. Solving only one of the problems in these cases may not necessarily eliminate such misses. For example, if a miss occurs due to both false sharing and capacity problems, then eliminating the false sharing problem by reducing block size will likely not eliminate that miss. On the other hand, sharing misses are in a sense more fundamental than capacity misses since they will remain even if the size of cache is increased to infinity, so we give them priority in the classification of multiple-cause misses. All misses with true sharing in their names in the resulting classification are called *essential coherence misses*. They would occur even with infinite caches, single-word blocks, and all data preloaded into all caches (i.e., no cold misses). Example 5.11 illustrates these definitions of miss categories.

EXAMPLE 5.11 Suppose three processors, P_1, P_2, and P_3, issue the memory operations shown in the first few columns of Table 5.4 (the first column indicates virtual time or steps). Use the miss classification algorithm to classify the misses in the last column. Assume that each processor's cache consists of only a single four-word cache block and that all the caches are initially empty.

Answer The results are shown in Table 5.4. ■

Impact of Block Size on Miss Rate

Applying the classification algorithm of Figure 5.20 to simulated runs of a workload, we can determine how frequently the various kinds of misses occur in programs and how the frequencies change with variations in cache organization, such as block size. Figure 5.21 shows the decomposition of the misses for the example applications running on 16 processors, with 1-MB four-way set-associative caches each, as the cache block size is varied from 8 bytes to 256 bytes. The bars show the four basic types of misses: cold misses (cases 1 and 2), capacity—including conflict—misses (case 9), true sharing misses, (cases 4, 6, 8, 10, 12), and false sharing misses (cases 3, 5, 7, and 11). In addition, they show the frequency of *upgrades*—writes that find the block in the cache but in the shared state. Upgrades are different from the other types of misses since the cache already has the valid data and only needs exclusive ownership. While they are not included in the classification scheme of Figure 5.20, they are still usually considered to be misses since they generate traffic on the interconnect and can stall the processor.

For each individual application, the miss characteristics change with block size much as we would expect from our understanding of the program and the miss categories. Cold, capacity, and true sharing misses tend to decrease with increasing block size because the additional data brought in with each miss is accessed before the block is replaced, due to spatial locality. However, false sharing misses tend to increase with block size. In all cases, true sharing is a significant fraction of the misses, so even with ideal, infinite caches, the miss rate and bus bandwidth will not go to zero. However, the overall characteristics differ widely across programs. For example, the size of the true sharing component varies significantly. Some applica-

Table 5.4 Classifying Misses in an Example Reference Stream from Three Processors

Time	P_1	P_2	P_3	Miss Classification
1	ld w0		ld w2	P_1 and P_3 miss; but we will classify later on replace/inval
2			st w2	$P_{1.1}$: pure cold miss; $P_{3.2}$: upgrade
3		ld w1		P_2 misses, but we will classify later on replace/inval
4		ld w2	ld w7	P_2 hits; P_3 misses; $P_{3.1}$: cold miss
5	ld w5			P_1 misses
6		ld w6		P_2 misses; $P_{2.3}$: cold true sharing miss (w2 accessed)
7		st w6		$P_{1.5}$: cold miss; $P_{2.7}$: upgrade; $P_{3.4}$: pure cold miss
8	ld w5			P_1 misses
9	ld w6		ld w2	P_1 hits; P_3 misses
10	ld w2	ld w1		P_1, P_2 miss; $P_{1.8}$: pure true share miss; $P_{2.6}$: cold miss
11	st w5			P_1 misses; $P_{1.10}$: pure true sharing miss
12			st w2	$P_{2.10}$: capacity miss; $P_{3.11}$: upgrade
13			ld w7	P_3 misses; $P_{3.9}$: capacity miss
14			ld w2	P_3 misses; $P_{3.13}$: inval cap false sharing miss
15	ld w0			P_1 misses; $P_{1.11}$: capacity miss

If multiple references are listed in the same row, we assume that P_1 issues before P_2 and P_2 issues before P_3. The notation ld/st w*i* refers to load/store of word *i*. W1 through w4 are on the same cache block, and so on. The notation $P_{i,j}$ points to the memory reference issued by processor *i* at row *j*.

tions show a substantial increase in false sharing with block size, whereas others show almost none. Furthermore, the figure shows data only for the default data sets. In practice it is very important to examine the results as the input data set size and number of processors are scaled before drawing conclusions about the false sharing or spatial locality of an application (see Chapter 4). Let us investigate the properties of the applications that give rise to differences in miss characteristics observed at the machine level and that allow us to understand scaling qualitatively.

Relation to Application Structure

Multiword cache blocks exploit spatial locality by prefetching data surrounding the accessed address. Of course, beyond a point, larger cache blocks can hurt performance by (1) prefetching unneeded data, (2) causing increased conflict misses as the number of distinct blocks that can be stored in a finite cache decreases with increasing block size, and (3) causing increased false sharing misses. Spatial locality in parallel programs tends to be lower than in sequential programs because, when a

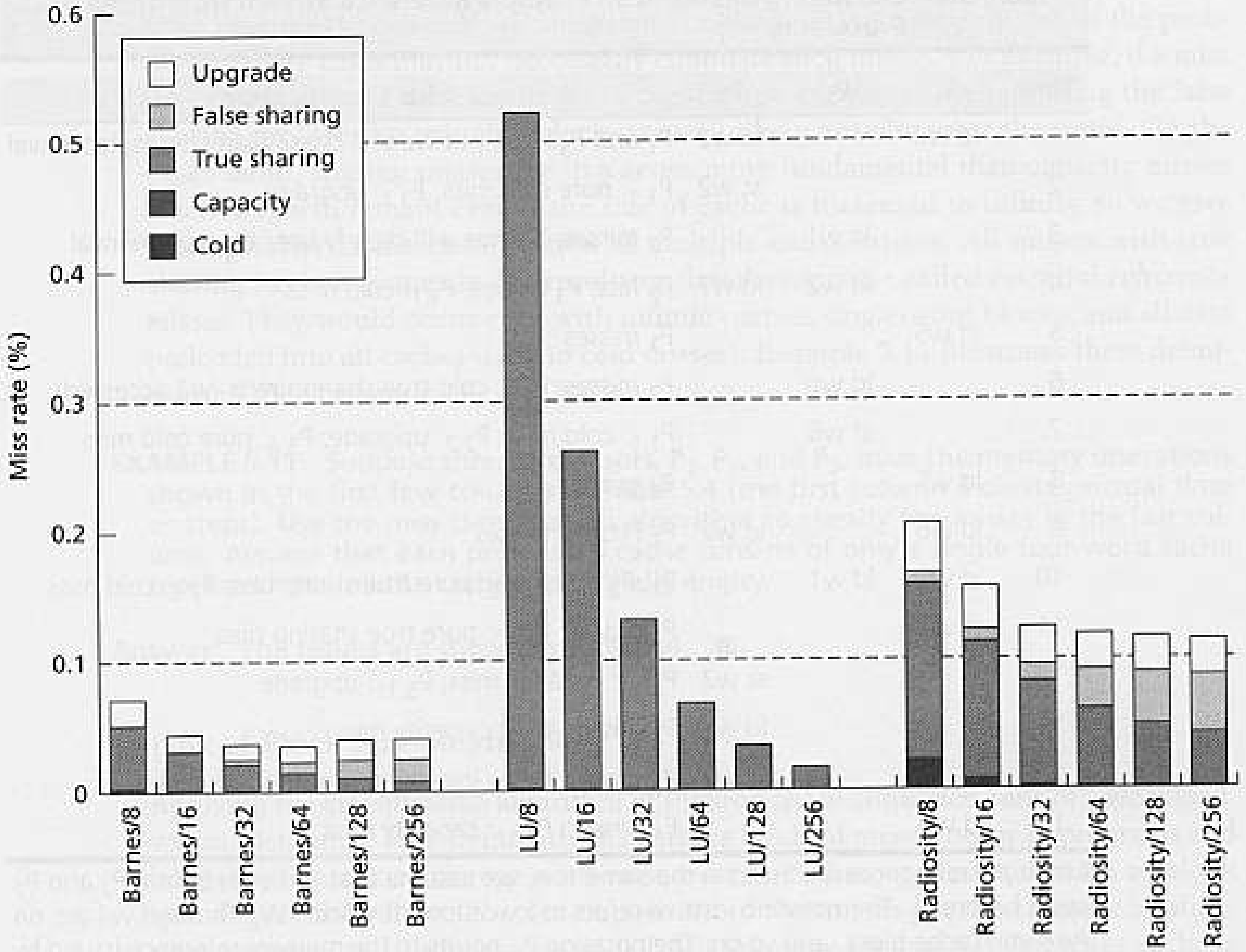


FIGURE 5.21(a) Breakdown of application miss rates as a function of cache block size for 1-MB caches per processor for Barnes-Hut, LU, and Radiosity applications. Conflict misses are included in capacity misses. The breakdown and behavior of misses vary greatly across applications, but we can observe some common trends. Cold misses and capacity misses tend to decrease quite quickly with block size as a result of spatial locality. True sharing misses also tend to decrease, whereas false sharing misses increase. While the false sharing component is usually small for small block sizes, it sometimes remains small and sometimes increases very quickly. Upgrades are shown at the top of the bars and without shading, so they can be ignored if desired.

memory block is brought into the cache, some of the data therein may belong to another processor and will not be used by the processor performing the miss. As an extreme example, some parallel programs assign adjacent elements of an array to different processors in order to ensure good load balance and in the process substantially decrease the spatial locality of the program.

The data in Figure 5.21 shows that LU and Ocean have good spatial locality and little false sharing even in the parallel case. The miss rates for many components drop proportionately to increases in cache block size, and false sharing misses are essentially nonexistent. This is in large part because these array-based codes use

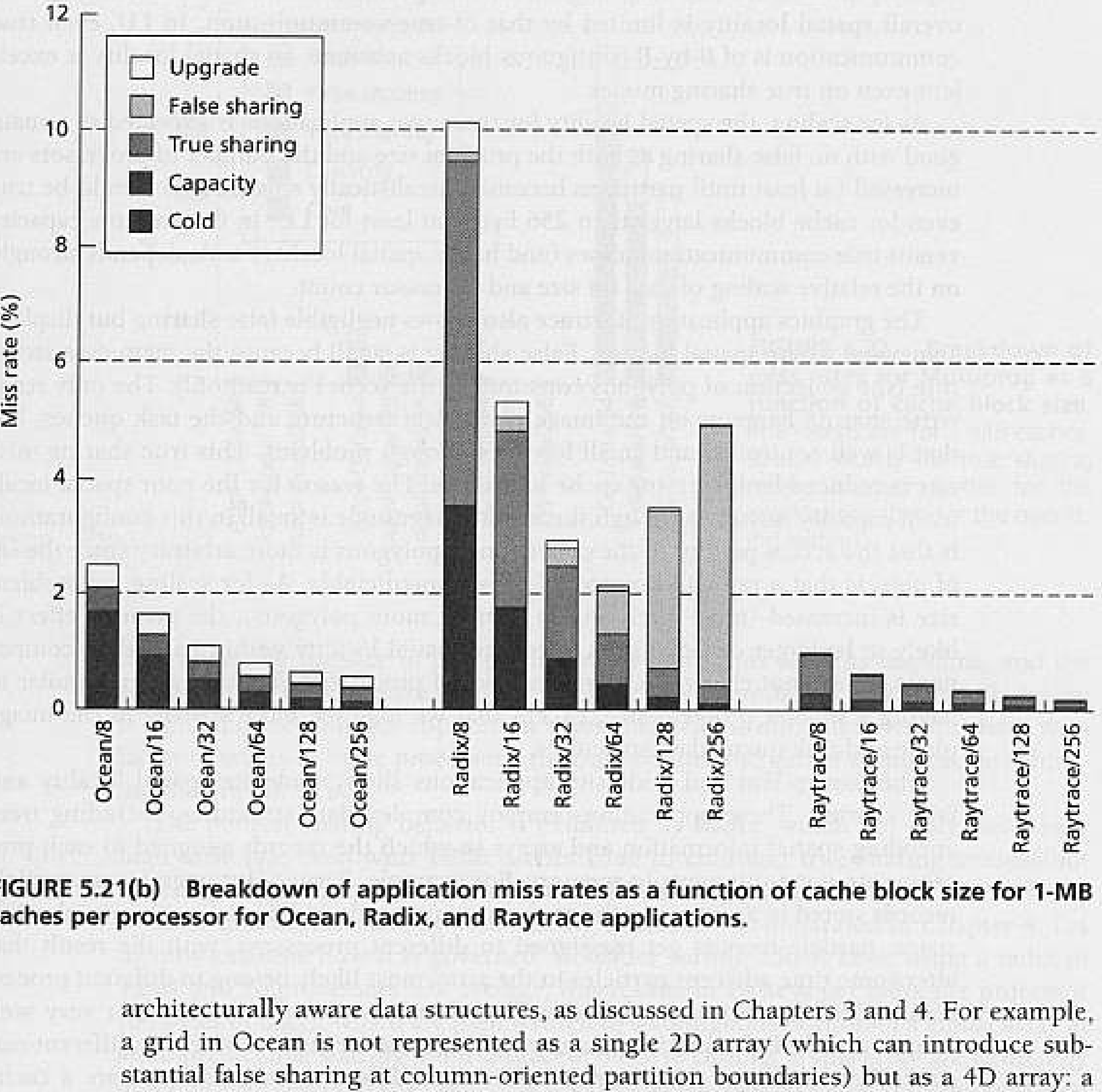


FIGURE 5.21(b) Breakdown of application miss rates as a function of cache block size for 1-MB caches per processor for Ocean, Radix, and Raytrace applications.

architecturally aware data structures, as discussed in Chapters 3 and 4. For example, a grid in Ocean is not represented as a single 2D array (which can introduce substantial false sharing at column-oriented partition boundaries) but as a 4D array: a 2D array of blocks, each of which is itself a 2D array. Such structuring, by programmers or compilers, ensures that most accesses are unit stride and over substantial, contiguous blocks of data, thus the nice behavior.

In Ocean, capacity misses are significant, but they are to the interior elements of a process's partition, so they have very good spatial locality. One difference with LU is that true sharing misses in Ocean do not exhibit such good spatial locality. Most of the true sharing misses are to elements at the borders of neighboring partitions. These exhibit good spatial locality at row-oriented borders where the data to be fetched is contiguous in the address space. However, when a processor accesses an element at a column-oriented border, it fetches an entire cache block of interior elements of its neighbor's partition, which it will not use and therefore wastes. Since capacity misses are not very large with this problem and machine configuration, overall spatial locality is limited by that of true communication. In LU, even true communication is of *B*-by-*B* contiguous blocks at a time, so spatial locality is excellent even on true sharing misses.

As for scaling, the spatial locality for these two applications is expected to remain good with no false sharing as both the problem size and the number of processors are increased (at least until partitions become unrealistically small). This should be true even for cache blocks larger than 256 bytes, at least for LU. In Ocean, how capacity versus true communication misses (and hence spatial locality) scale depends strongly on the relative scaling of data set size and processor count.

The graphics application Raytrace also shows negligible false sharing but displays somewhat worse spatial locality. False sharing is small because the main data structure (the collection of polygons constituting the scene) is read-only. The only read-write sharing happens on the image plane data structure and the task queues, but that is well controlled and small for large enough problems. This true sharing miss rate is reduced by increasing cache block size. The reason for the poor spatial locality of capacity misses (although the overall magnitude is small in this configuration) is that the access pattern to the collection of polygons is quite arbitrary since the set of objects that a ray will bounce off of is unpredictable. As for scaling, as problem size is increased (most likely in the form of more polygons), the primary effect is likely to be larger capacity miss rates; the spatial locality within individual components should not change. A larger number of processors is in many ways similar to having a smaller problem size, except that we may see more sharing in the image plane and task queue data structures.

The Barnes-Hut and Radiosity applications show moderate spatial locality and false sharing. These applications employ complex data structures, including trees encoding spatial information and arrays in which the records assigned to each processor are not contiguous in memory. For example, Barnes-Hut operates on particle records stored in an array. As the application proceeds and particles move in physical space, particle records get reassigned to different processors, with the result that after some time adjacent particles in the array most likely belong to different processors. Spatial locality is exploited well within a particle record but not very well across records. False sharing becomes a problem at large block sizes for different reasons. First, different processors may write to different records that share a cache block. Second, a particle data structure (record) contains both fields that are being modified by the owner of that particle in a phase (e.g., the current force on this particle in the force calculation phase) and fields that are read by other processors and are not being modified in this phase (e.g., the current position of the particle). Since these two fields may fall in the same cache block for large block sizes, false sharing results. It is possible to eliminate such false sharing by splitting the particle data structure according to the access patterns of the fields, but that is not done in this program since the absolute magnitude of the miss rate is small. As problem size and the number of processors are scaled, the miss rate behavior of Barnes-Hut is expected to change little. This is because the working set size changes very slowly (as the log of the number of particles, unlike Ocean and Raytrace), spatial locality is

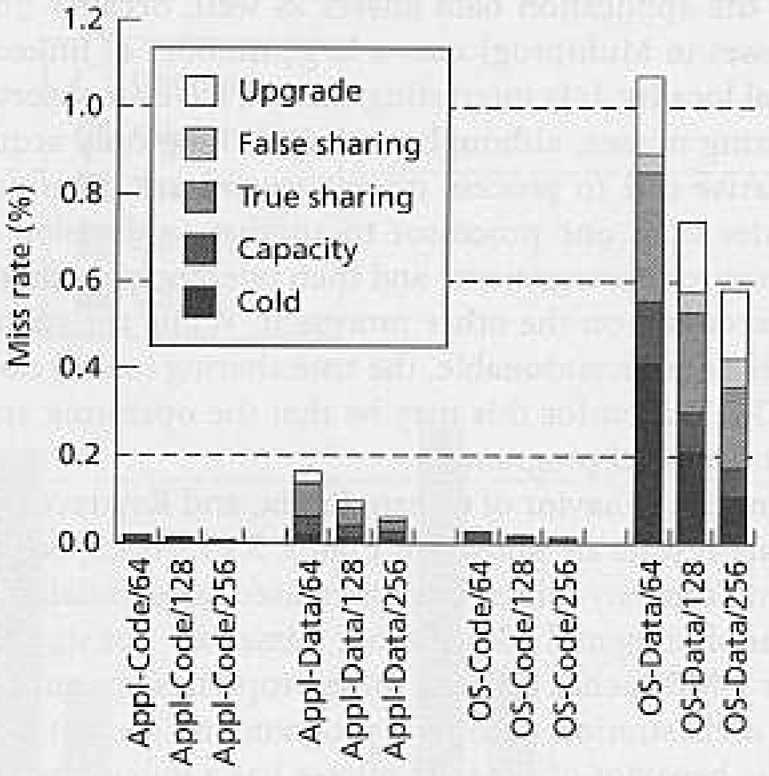


FIGURE 5.22 Breakdown of miss rates for Multiprog as a function of cache block size. The results are for 1-MB caches. Spatial locality for true sharing misses is much better for the applications than for the operating system.

determined by the size of one particle record and thus remains the same, and the sources of false sharing are not very sensitive to the number of processors. Radiosity is a much more complex application whose behavior is difficult to reason about with larger data sets or more processors; the only option is to gather empirical data showing the growth trends.

The poorest sharing behavior is exhibited by Radix, which not only has a very high miss rate even with 1-MB caches (due to cold and true sharing misses) but which gets significantly worse due to false sharing misses for block sizes of 128 bytes or more. The effect of false sharing in Radix was illustrated in Chapter 4. Let us now examine how it is governed. Consider sorting 256-K keys, using a radix of 1,024 and 16 processors. On average, this results in 16 keys per radix per processor (64 bytes of data), which are then written to a contiguous portion of a global array at an unpredictable starting point. Adjacent 64-byte chunks in this array are written by different processors. If the cache block size is larger than 64 bytes, the high potential for false sharing is clear. As the problem size is increased we will clearly see much less false sharing. The effect of increasing the number of processors is exactly the opposite. Radix illustrates quite dramatically that it is not sufficient to look at a given problem size and number of processors and, based on that, draw conclusions of whether or not false sharing or spatial locality is a problem. It is very important to understand how the results are dependent on the key parameters chosen in the experiment and how these parameters may vary in reality.

Data for the Multiprog workload for 1-MB caches is shown in Figure 5.22. The data is shown separately for user code, user data, kernel code, and kernel data. For code, there are only cold and capacity misses. Furthermore, we see that the spatial locality in operating system data references is not very good. This is true, to a somewhat lesser extent, for the application data misses as well, because gcc (the main application causing misses in Multiprog) uses a large number of linked lists, which do not offer good spatial locality. It is interesting that we have an observable fraction of application true sharing misses, although we are running only sequential applications. These misses arise due to process migration and are incurred when a sequential process migrates from one processor to another (a decision made by the operating system for resource management) and then references memory blocks that it wrote while it was executing on the other processor. While the spatial locality in cold and capacity misses is quite reasonable, the true sharing misses do not decrease at all for kernel data. One reason for this may be that the operating system has not been well structured as a parallel program.

Finally, let us examine the behavior of Ocean, Radix, and Raytrace for smaller 64-KB caches. The miss rate results are shown in Figure 5.23. As expected, the overall miss rates are higher, and capacity misses have increased substantially. The effects of cache block size for true sharing and false sharing misses are not significantly different from the results for 1-MB caches because these properties are quite fundamental to the assignment and orchestration used by a program and are not too sensitive to cache size. However, the behavior of capacity misses has a much larger effect on the behavior of the overall miss rate. For example, in Ocean, capacity misses now dominate sharing misses; since they have much better spatial locality, the overall miss rate decreases much more quickly with increasing block size than it did with 1-MB caches. (Very large blocks in a small cache can have the problem that blocks may be replaced from the cache due to conflicts before the processor has had a chance to reference all of the words in them.) In Raytrace, capacity misses have somewhat worse spatial locality than true sharing misses, so the overall benefits of large blocks look worse with smaller caches. Results for false sharing and spatial locality for other applications can be found in the literature (Torrellas, Lam, and Hennessy 1994; Jeremiassen and Eggers 1991; Woo et al. 1995).

While larger cache blocks reduce the miss rate for most of our applications, within the range of block sizes we consider they have two important potential disadvantages. First, they can increase the cost of each miss since more data has to be transferred across the bus (although techniques like only waiting for the referenced word to arrive before allowing the processor to proceed, called a *critical word restart* approach, can alleviate this). Second, they increase traffic, and hence contention, if the whole block is not useful.

Impact of Block Size on Bus Traffic

Let us briefly examine the impact of cache block size on bus traffic rather than miss rate. While the number of misses and total traffic generated are clearly related, their impact on observed performance can be quite different. Misses have a cost that may contribute directly to performance, even though modern microprocessors try hard to hide the latency of misses by overlapping it with other activities. Traffic, on the

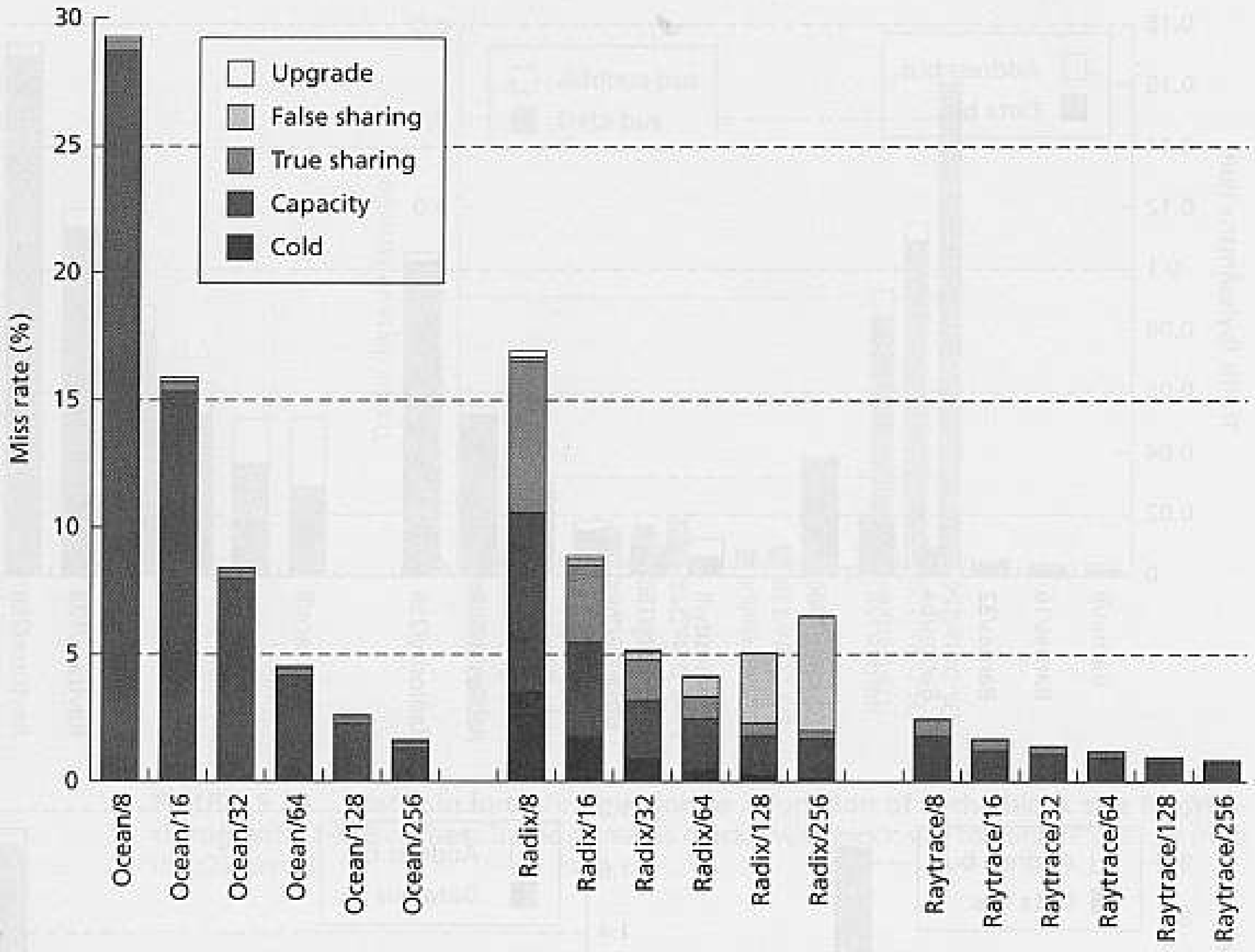
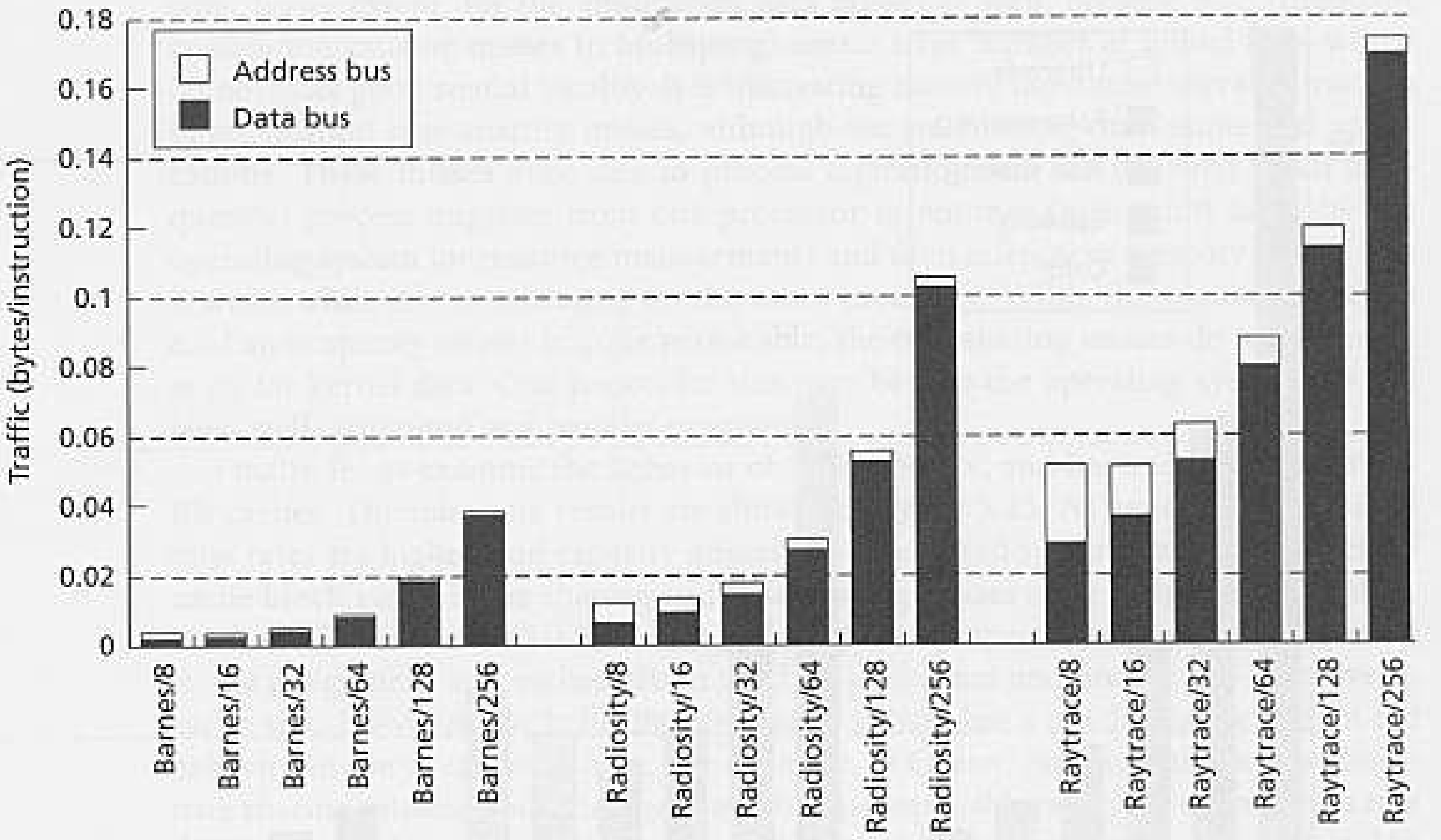


FIGURE 5.23 Breakdown of application miss rates as a function of cache block size for 64-KB caches. Capacity misses are now a much larger fraction of the overall miss rate. Capacity miss rates decrease differently with block size for different applications.

other hand, affects performance indirectly by causing contention and hence increasing the cost of other misses. For example, if an application program's misses are reduced significantly by increasing the cache block size, but the bus traffic is increased by 50%, this might be a reasonable trade-off if the application was originally using only 10% of the available bus and memory bandwidth. Increasing the bus and memory utilization to 15% is unlikely to increase the miss latencies significantly. However, if the application was originally using 75% of the bus and memory bandwidth, then increasing the block size is probably a bad idea.

Figure 5.24 shows the total bus traffic for our applications in bytes/instruction or bytes/FLOP as the cache block size is varied. Three key points can be observed from this graph. First, traffic behaves very differently than miss rate. Only LU shows monotonically decreasing total traffic for the block sizes used. Most other applications see a doubling or tripling of traffic as block size becomes large. Second, the



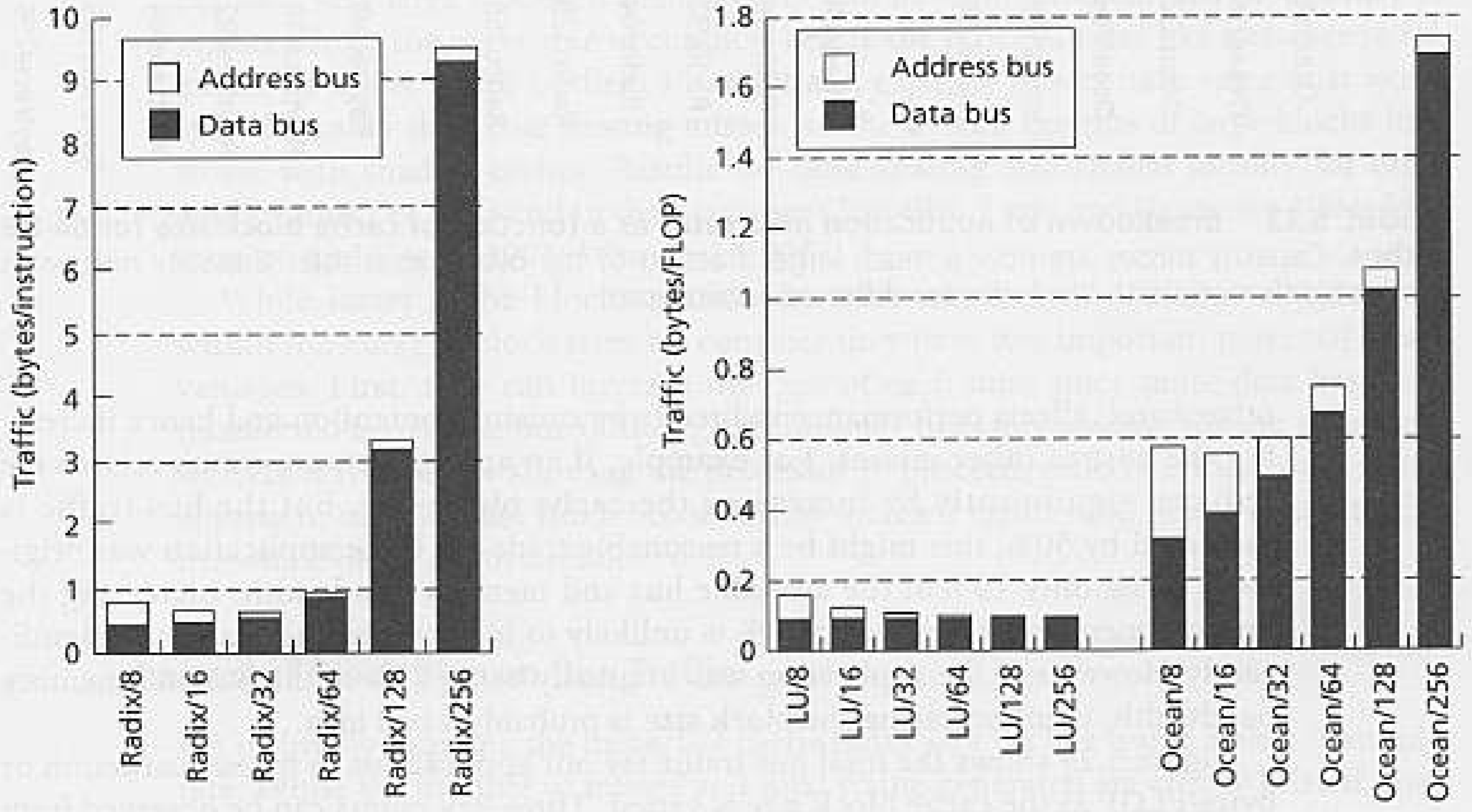


FIGURE 5.24 Traffic (in bytes/instruction or bytes/FLOP) as a function of cache block size with 1-MB caches per processor. Data traffic increases quite quickly with block size when communication misses dominate, except for applications like LU that have excellent spatial locality on all types of misses. Address (including command) bus traffic tends to decrease with block size since the miss rate and, hence, number of blocks transferred decrease.

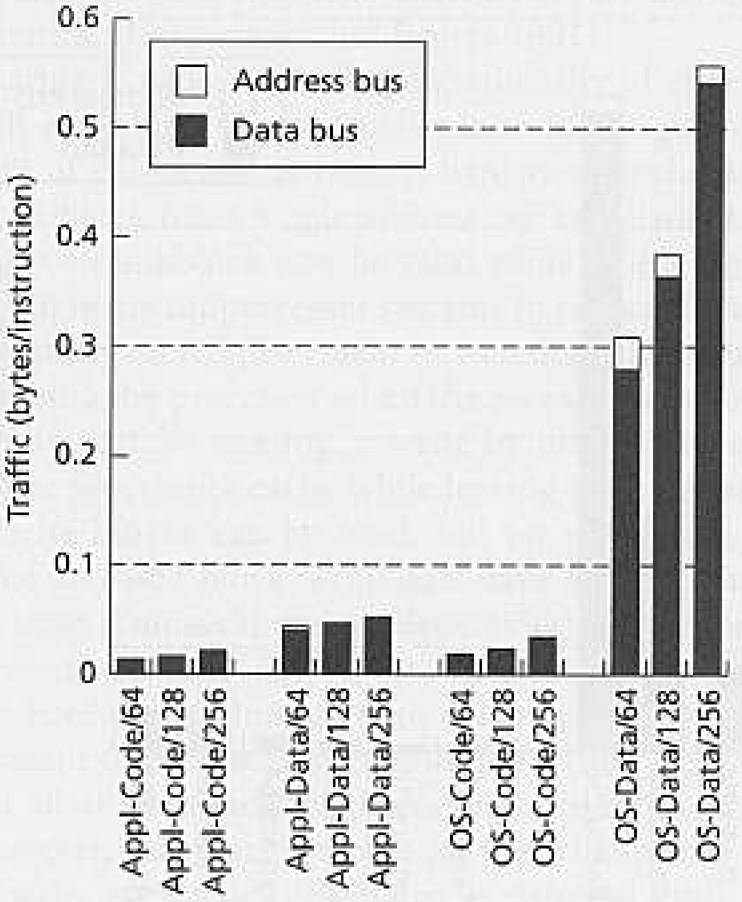


FIGURE 5.25 Traffic in bytes/instruction as a function of cache block size for Multiprog with 1-MB caches. Traffic increases quickly with block size for data references from the OS kernel.

overall traffic requirements for the applications are still small, even for 256-byte block sizes, with the exception of Radix. Radix's large bandwidth requirements (approximately 650 MB/s per processor for 128-byte cache blocks, assuming a sustained 200-MIPS processor) reflect its false sharing problems at large block sizes. Third, the constant address and command traffic overhead for each bus transaction or miss comprises a significant fraction of total traffic for small block sizes. Hence, although actual application data traffic usually increases as we increase the block size due to poor spatial locality, the total traffic is often minimized at 16–32 bytes rather than 8 bytes due to the amortization of the overhead with improved miss rates.

Figure 5.25 shows the traffic data for Multiprog. While the increase in traffic from 64-byte cache blocks to 128-byte blocks is small, the jump at 256-byte blocks is much more substantial (primarily due to kernel data references). Finally, Figure 5.26 shows the traffic results for 64-KB caches for the three relevant applications. For Ocean, even 64- and 128-byte cache blocks don't look so bad, due to the dominance of capacity misses that have good spatial locality.

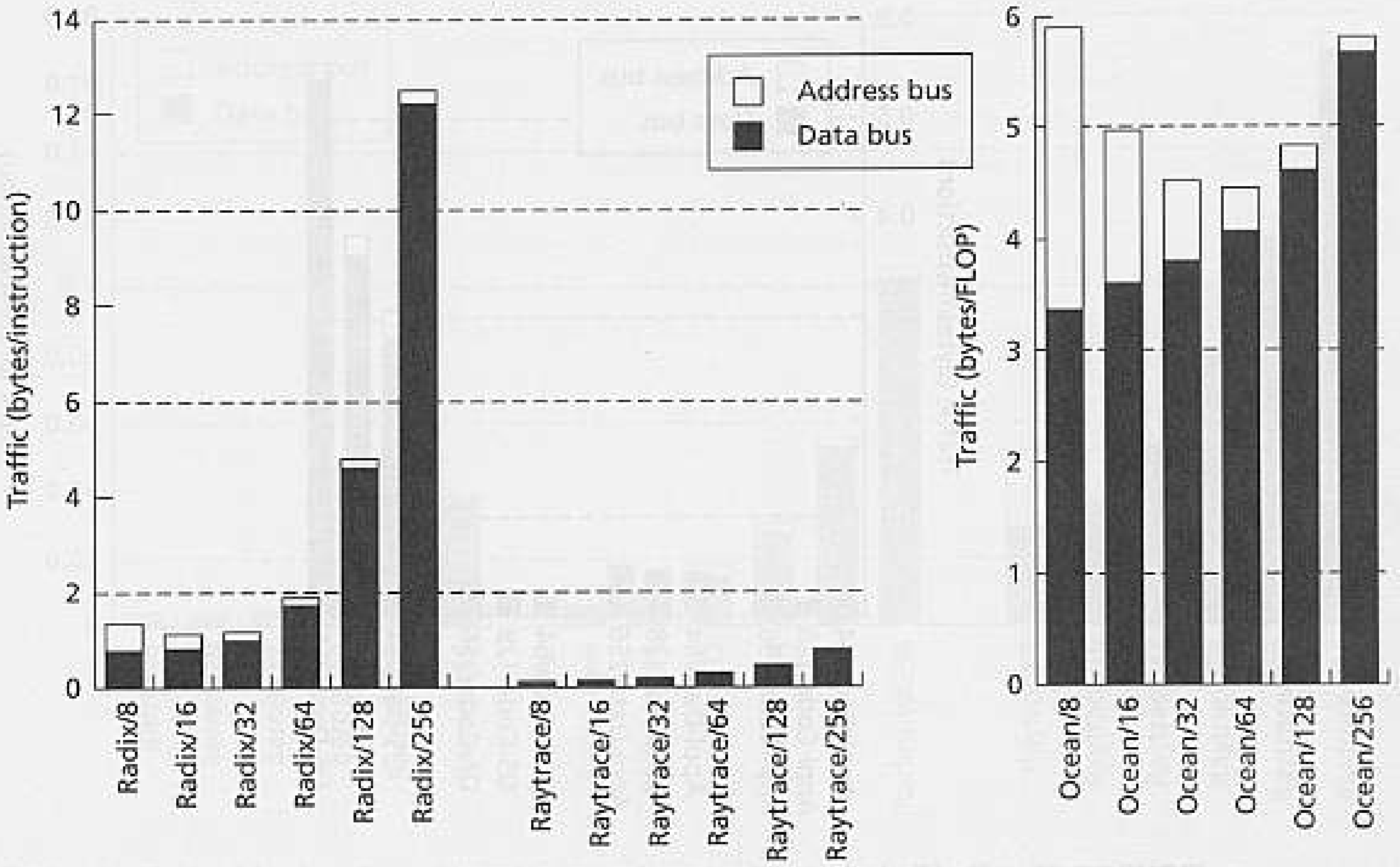


FIGURE 5.26 Traffic (in bytes/instruction or bytes/FLOP) as a function of cache block size with 64-KB caches per processor. Traffic increases more slowly now for Ocean than with 1-MB caches since the capacity misses that now dominate exhibit excellent spatial locality (traversal of a process's assigned subgrid). However, traffic in Radix increases quickly once the threshold block size that causes false sharing is exceeded.

Alleviating the Drawbacks of Large Cache Blocks

The trend toward larger cache block sizes is driven by the increasing gap between processor performance and memory access time. The larger block size amortizes the cost of the bus transaction and memory access across a greater amount of data. The increasing density of processor and memory chips makes it possible to employ large first-level and second-level caches so that the prefetching of data obtained through a larger block size dominates the small increase in conflict misses. However, this trend may bode poorly for multiprocessor designs because false sharing becomes a larger problem. Fortunately, hardware and software mechanisms can be employed to counter the effects of large block size.

Software techniques to reduce false sharing and improve locality on coherence misses are discussed in detail later in the chapter. They essentially involve organizing data structures or work assignments so that data accessed by different processes is not interleaved finely in the shared address space. One example is the use of higher-dimensional arrays so blocks or partitions are wholly contiguous. Compiler techniques have also been developed to automate some methods of laying out data to reduce false sharing (Jeremiassen and Eggers 1991).

Since false sharing is caused by a large granularity of coherence, the way to reduce it while still exploiting spatial locality is to use large blocks for data transfer but a smaller unit of coherence. A natural hardware mechanism is the use of subblocks. Each cache block has a single address tag but distinct state bits for each of several subblocks. One subblock may be valid while others are invalid or dirty. This technique is used in many uniprocessor systems to reduce the amount of data that is copied back to memory on a replacement or to reduce the memory access time on a read miss by resuming the processor when the accessed subblock is present (critical word restart). To avoid false sharing, a write by one processor may invalidate the subblock in another processor's cache while leaving the other subblocks valid. Alternatively, small cache blocks can be used, but on a miss the system can prefetch blocks beyond the accessed block. Proposals have also been made for caches with adjustable block sizes (Dubnicki and LeBlanc 1992). The disadvantage of these approaches is increased state and complexity beyond a commodity cache design.

A more subtle hardware technique is to delay propagating or applying invalidations from a processor until it has issued multiple writes. Delaying invalidations and performing them all at once reduces the occurrence of intervening read misses to those blocks. However, this sort of technique can change the memory consistency model in subtle ways, so further discussion is deferred until Chapter 9 where we consider weaker consistency models in the context of scalable machines. Another hardware technique to reduce false sharing is the use of update- rather than invalidation-based protocols.

5.4.5 Update-Based versus Invalidation-Based Protocols

Whether writes should cause other cached copies to be updated or invalidated has been the subject of considerable debate. Various vendors have taken different stands and, in fact, have changed their position from one design to the next. The controversy arises because the relative performance of update-based versus invalidation-based protocols depends strongly on the sharing patterns exhibited by the workload and on the cost of various underlying operations. Intuitively, if the processors that were using the data before it was updated (written) are likely to want to see the new values in the future, updates should perform better than invalidations. However, if the processors holding the old data are never going to use it again, the update traffic is useless and just consumes interconnect and controller resources. Invalidations would clean out the old copies and eliminate the apparent sharing. This "pack rat" phenomenon with update protocols is especially irritating under multiprogrammed use of a machine, when sequential processes migrate from processor to processor under OS control so that useless updates are performed in caches of processors that are no longer running that process. It is easy to construct cases in which either scheme does substantially better than the other, as illustrated by Example 5.12.

EXAMPLE 5.12 Consider the following two program reference patterns:

- *Pattern 1:* Repeat k times; processor 1 writes a new value into variable V and processors 2 through P read the value of V. This represents a one-producer-many-consumer scenario that may arise, for example, when processors are accessing a highly contended flag for one-to-many event synchronization.
- *Pattern 2:* Repeat k times; processor 1 writes M times to variable V and then processor 2 reads the value of V. This represents a sharing pattern that may occur between pairs of processors, where the first successfully computes and accumulates values into a variable and then when the accumulation is complete, another processor reads the value.

What is the relative cost for update- and invalidation-based protocols in terms of the number of cache misses and bus traffic? Assume that an invalidation/upgrade transaction consumes 6 bytes (5 bytes for address plus 1 byte for command), an update takes 14 bytes (6 bytes for address and command and 8 bytes of data for the updated word), and a regular cache miss takes 70 bytes (6 bytes for address and command plus 64 bytes of data corresponding to cache block size). Also assume that $P = 16$, $M = 10$, $k = 10$, and that all caches initially are empty.

Answer With an update scheme in pattern 1, the first iteration on all P processors will incur a regular cache miss (including processor 1 when it writes) plus an update due to the write. In subsequent $k - 1$ iterations, no more misses will occur and only one update per iteration will be generated. Thus, overall we will see misses = P = 16; traffic = $P \times$ RdMiss + $(k - 1) \times$ Update = $16 \times 70 + 10 \times 14$ = 1,260 bytes.

With an invalidate scheme, all P processors will incur a regular cache miss in the first iteration. In subsequent $k - 1$ iterations, processor 1 will generate an upgrade, but all others will experience a read miss. Thus, counting upgrades as misses, overall we will see misses = $P + (k - 1) \times P = 16 + 9 \times 16 = 160$, of which 151 are read misses and 9 are upgrades; traffic = read misses × RdMiss + $(k - 1) \times$ Upgrade = $151 \times 70 + 9 \times 6$ = 10,624 bytes.

With an update scheme on pattern 2, the first iteration will incur two regular cache misses, one for processor 1 and the other for processor 2. In subsequent $k - 1$ iterations, no more misses will be generated, but M updates will be generated in each iteration. Thus, overall we will see misses = 2; traffic = $2 \times$ RdMiss + $M \times (k - 1) \times$ Update = $2 \times 70 + 10 \times 9 \times 14$ = 1,400 bytes.

With an invalidate scheme, two regular cache misses will occur in the first iteration. In subsequent $k - 1$ iterations, one upgrade (for the first write only) plus one regular read miss will be generated in each iteration. Thus, counting upgrades as misses, overall we will see misses = $2 + (k - 1) \times 2 = 2 + 9 = 11$; traffic = misses × RdMiss + $(k - 1) \times$ Upgrade = $11 \times 70 + 9 \times 6$ = 824 bytes. ■

These example patterns suggest that it might be possible to design schemes that capture the advantages of both update and invalidate protocols. The success of such schemes will depend on their costs and on the sharing patterns for real parallel programs and workloads. Let us briefly explore the design options and then employ workload-driven evaluation.

Combining Update- and Invalidation-Based Protocols

One way to take advantage of both update and invalidate protocols is to support both in hardware and to decide dynamically at page granularity whether coherence for a given page is to be maintained using an update or an invalidate protocol. The decision about the choice of protocol can be indicated by making a system call. The main advantage of such schemes is that they are relatively easy to support; they utilize the TLB to indicate to the rest of the coherence subsystem which of the two protocols to use. The main disadvantage of such schemes is the burden they put on the programmer to choose protocols for pages or data structures. The decision task is also made difficult because of the coarse granularity at which control is made available; data structures that desire different protocols may fall on the same page.

An alternative is to choose the protocol at a cache block granularity, by observing the sharing behavior at run time. Ideally, for each write, we would like to be able to peer into the future references that will be made to that cache block by all processors and then decide whether to invalidate other copies or to do an update. Since this information is obviously not available, and since there are substantial perturbations due to cache replacements and false sharing, a more practical scheme is needed.

So-called competitive schemes change the protocol for a block between invalidate and update in hardware based on observed patterns at run time. The key attribute of such schemes is that if a wrong decision is made once for a cache block, the losses due to that wrong decision should be kept bounded and small (Karlin et al. 1986). For instance, if a block is currently using update mode, it should not remain in that mode if one processor is continuously writing to it but none of the other processors are reading values from it.

One class of schemes that has been proposed to bound the losses of update protocols works as follows (Grahn, Stenstrom, and Dubois 1995). Starting with the base Dragon update protocol described in Section 5.3.3, associate a countdown counter with each block. Whenever a cache block is accessed by the local processor, the counter value for that block is reset to a threshold value *k*. Every time an update is received for a block, the counter is decremented. If the counter goes to zero, the block is locally invalidated. The consequence of the local invalidations is that the next time an update is generated on the bus, it may find that no other cache has a valid copy; in that case, that block will switch to the modified state (as per the Dragon protocol) and will stop generating updates. If some other processor now accesses that block, the block will again switch to shared state and this mixed protocol will again start generating updates.

A related approach implemented in the Sun SparcCenter 2000 is to selectively invalidate rather than update with some probability that is a parameter set when configuring the machine (Catanzaro 1997). Other mixed approaches may also be used. For example, one approach uses an invalidation-based protocol for first-level caches and, by default, an update-based protocol for second-level caches. However, if the L_2 cache receives a second update for the block while the block in the L_1 cache is still invalid, then the block is invalidated in the L_2 cache as well. When the block is thus invalidated in all other L_2 caches, writes to the block no longer cause updates.

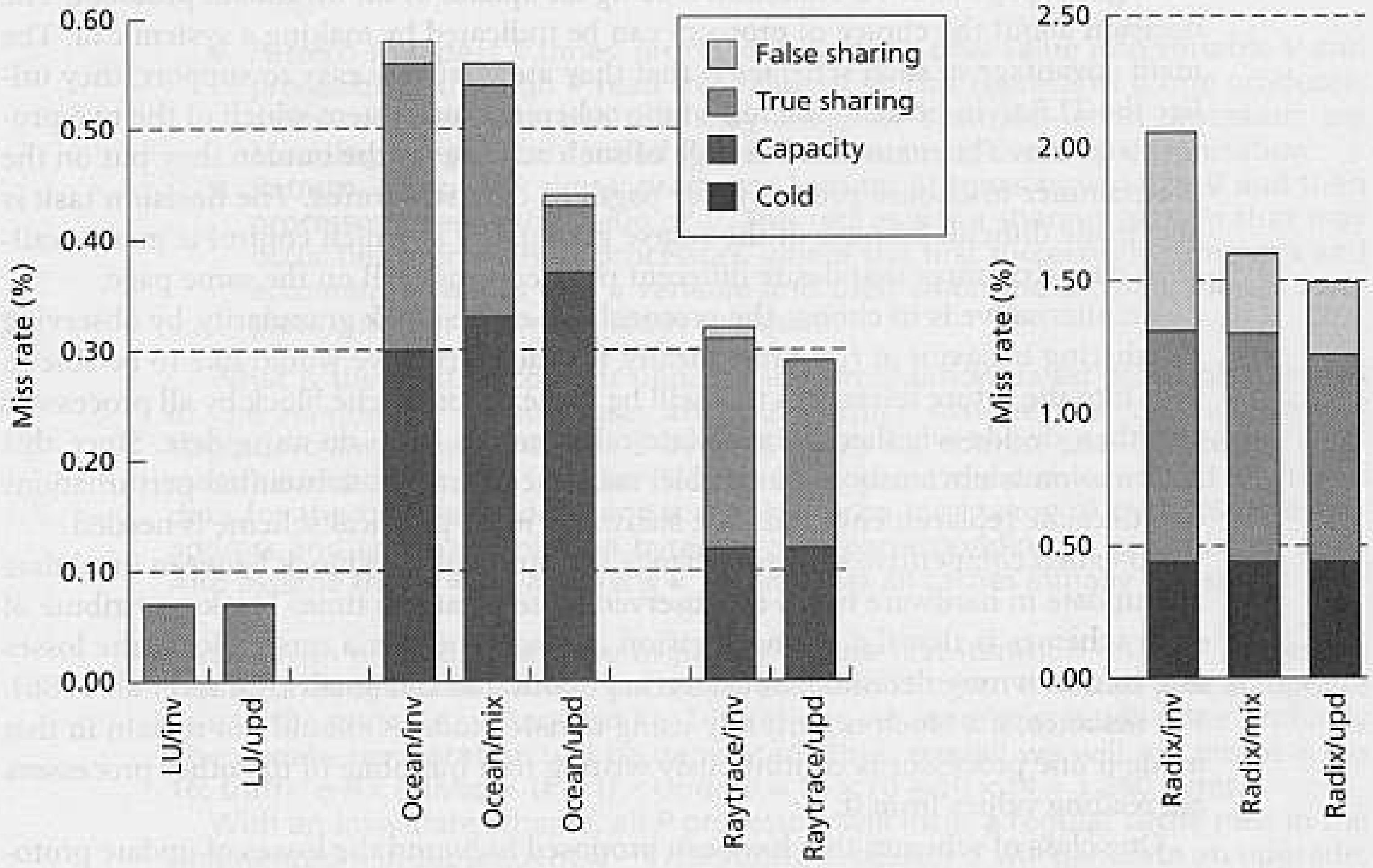


FIGURE 5.27 Miss rates and their decomposition for invalidate, update, and hybrid protocols. The data assumes 1-MB caches, 64-byte cache blocks, four-way set associativity, and threshold $k = 4$ for hybrid protocol.

Workload-Driven Evaluation

To assess the trade-offs among invalidate, update, and the mixed protocols just described, Figure 5.27 shows the miss rates by category for four applications using the default 1-MB four-way set-associative caches with a 64-byte block size. The mixed protocol used is the threshold-based scheme just described. We see that for applications with significant capacity miss rates, the misses sometimes increase with an update protocol. This makes sense because the protocol (with LRU replacement in a set) keeps data in processor caches that would have been removed by an invalidation protocol. For applications with significant true sharing or false sharing miss rates, these categories decrease with an update protocol: after a write update, the other caches holding the blocks can access them without a miss. Overall, the update protocol appears to be advantageous for the sum of these three categories and the mixed protocol falls in between. The category that is not shown in this figure, however, is the upgrade or update operations for these protocols. This data is presented in Figure 5.28. Note that the scale of the graphs has changed because update operations are roughly four times more prevalent than misses. It is useful to separate these operations from other misses because the way they are handled in the machine is

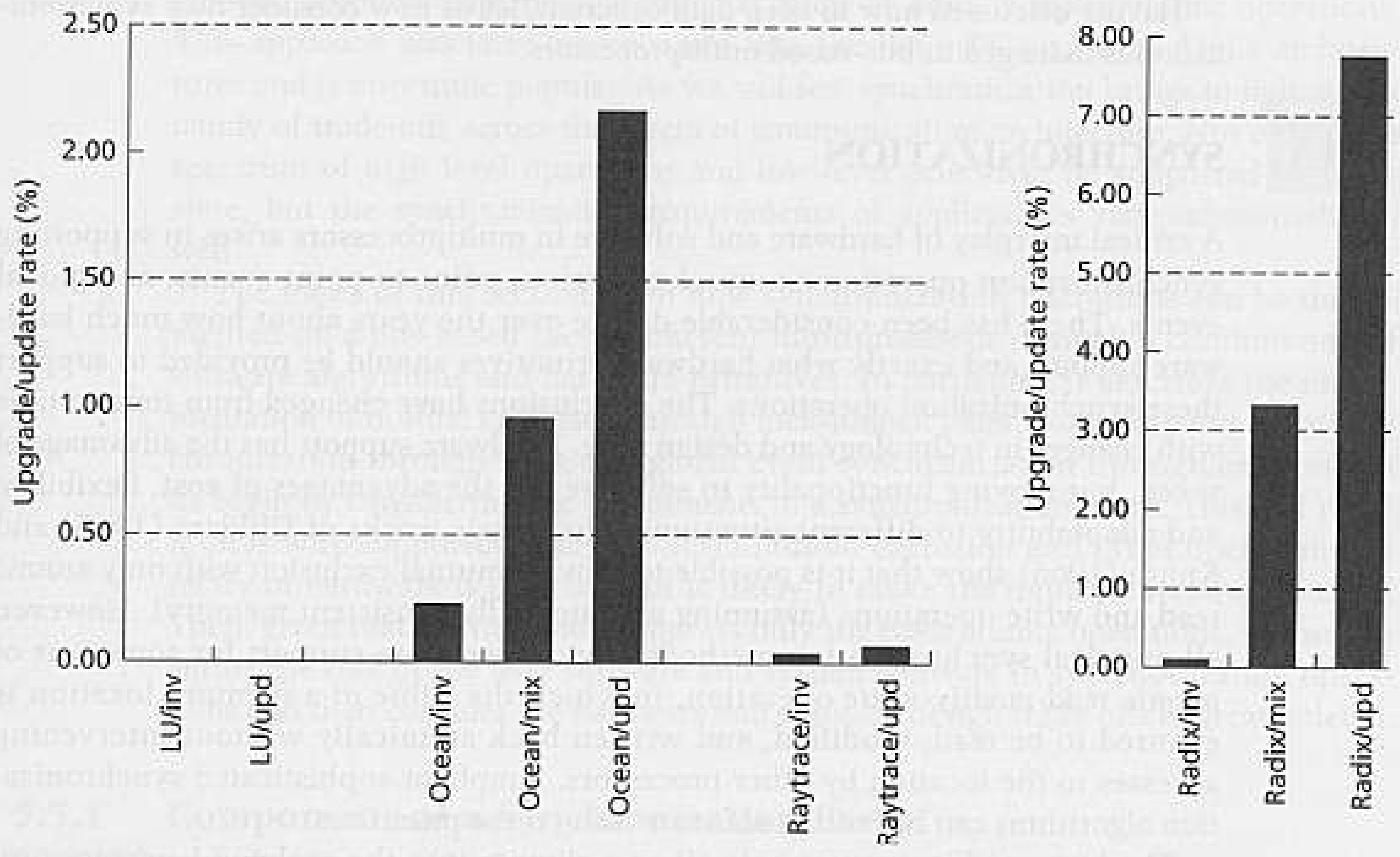


FIGURE 5.28 Upgrade and update rates for invalidate, update, and mixed protocols. The data assumes 1-MB caches, 64-byte cache blocks, four-way set associativity, and threshold $k = 4$ for hybrid protocol. Rates are measured relative to total memory references.

likely to be different. Updates are a single-word write rather than a full cache block transfer. Because the data is being pushed from where it is being produced, it may arrive at the consumer before it is needed. Even for the producer, the latency of update and upgrade operations may be less critical than that of misses since it is quite easily hidden from the processor's critical path (see Chapter 11).

Unfortunately, the traffic associated with updates is quite substantial. In large part, this occurs because multiple writes are made by a processor to the same block before a read, all generating updates. With the invalidate protocol, the first of these writes may cause an invalidation, but the rest can simply accumulate locally in the block and be transferred in one bus transaction on a flush or a write back (see Example 5.12). The increased traffic causes contention and can greatly increase the cost of misses. Sophisticated update schemes might attempt to delay the update to achieve a similar effect (by merging writes in the write buffer) or use other techniques to reduce traffic and improve performance (Dahlgren 1995). However, the increased bandwidth demand, the complexity of supporting updates, the trend toward larger cache blocks, and the pack rat phenomenon with the important case of multiprogrammed sequential workloads underlie the trend away from update-based protocols in the industry. We see in Chapter 8 that update protocols also have some other problems for scalable cache-coherent architectures, making it less attractive for microprocessors to support these protocols.

Having discussed how to keep data coherent, let us now consider how synchronization is managed in bus-based multiprocessors.

5.5 SYNCHRONIZATION

A critical interplay of hardware and software in multiprocessors arises in supporting synchronization operations: mutual exclusion, point-to-point events, and global events. There has been considerable debate over the years about how much hardware support and exactly what hardware primitives should be provided to support these synchronization operations. The conclusions have changed from time to time with changes in technology and design style. Hardware support has the advantage of speed, but moving functionality to software has the advantages of cost, flexibility, and adaptability to different situations. The classic works of Dijkstra (1965) and Knuth (1966) show that it is possible to provide mutual exclusion with only atomic read and write operations (assuming a sequentially consistent memory). However, all practical synchronization methods rely on hardware support for some sort of *atomic read-modify-write* operation, in which the value of a memory location is ensured to be read, modified, and written back atomically without intervening accesses to the location by other processors. Simple or sophisticated synchronization algorithms can be built in software using these primitives.

The history of instruction sets offers a glimpse into the evolving hardware support for synchronization. One of the key instruction set enhancements in the IBM 370 was the inclusion of a sophisticated atomic instruction, the *compare&swap* instruction, to support synchronization in concurrent programming on uniprocessor or multiprocessor systems. The compare&swap compares the value in a memory location with the value in a specified register and, if they are equal, swaps the value in the memory location with the value in a second specified register. The Intel x86 allows any instruction to be prefixed with a lock modifier to make it atomic; since the source and destination operands are memory locations, much of the instruction set can be used to implement various atomic operations involving even more than one memory location. Advocates of high-level language architecture have proposed that the user-level synchronization operations, such as locks and barriers, should be supported directly at the machine level, not just atomic read-modify-write primitives; that is, the synchronization "algorithm" itself should be implemented in hardware. This issue became very active during the reduced instruction set debates since the operations that access memory were scaled back to simple loads and stores with only one memory operand. The Sparc approach was to provide atomic operations involving a register or registers and a memory location using a simple swap (atomically swapping the contents of the specified register and memory location) and a compare&swap. MIPS left off atomic primitives in the early instruction sets, as did the IBM Power architecture used in the RS6000. The primitive that was eventually incorporated in MIPS was a novel combination of a special load and a conditional store, described later in this section, which allows a variety of higher-level read-modify-write operations to be constructed without requiring the design to implement them all. In essence, the pair of instructions can be used instead of a sin-