CACHE CONSIDERATIONS for MULTIPROCESSOR PROGRAMMERS

Mark D. Hill & James R. Larus

Although caches in most computers are invisible to programmers, they significantly affect program performance. This is particularly true for cache-coherent, shared-memory multiprocessors. This article presents recent research into the performance of parallel programs and its implications for programmers who may know little about caches.

In their quest for faster machines, computer architects design computers to exploit the locality present in most programs. Locality takes several forms: *Spatial locality* arises because two contemporaneous memory references are likely to access nearby words. *Temporal locality* occurs because a recently referenced memory word is likely to be accessed again. A *cache* maintains a copy of data in a local memory that requires less time to access than the original data. Typically, a cache is a high-speed buffer adjacent to a processor that holds copies of memory locations likely to be used soon [12]. Caches are managed by hardware to increase their speed. Caches favor programs in which most references are local and permit them to execute faster than similar programs with less locality. Most uniprocessor programs exhibit considerable locality, even if their programmer was unaware of the existence of caches.

Parallel computing introduces a new type of locality, called *processor*

locality, in which contemporaneous references to a memory word come from a single processor, rather than many different ones [1]. Many multiprocessors which devote substantial resources to providing a large cache for each processor to hide this feature, allowing programmers to write correct programs without reasoning about these caches. Although programmers find it not only possible, but easy to write multiprocessor programs in which each process has substantial locality, interactions among processes reduce performance and diminish the benefit of moving from a uniprocessor to a multiprocessor. This article describes the behavior of these hidden caches and presents some guidelines for programmers who wish to use them more effectively.

We are most interested in cache-coherent, shared-memory multiprocessors (*multis*) [5]. Many commercial multiprocessors, such as the Sequent Symmetry, Encore Multimax, and Alliant FX/8 are multis. Figure 1 shows the typical structure of these machines. Each processor contains a local cache that reduces the expected long delay of referencing main memory through an interconnection network (e.g., shared bus). As long as a processor accesses data that is not shared with any other processor, the cache works like a uniprocessor's cache, keeping a copy of recently used locations. However, when a memory location is shared among processors, a cache-coherence protocol ensures that each processor sees a consistent view of the datum, even though it may be stored in more than one cache [4]. These protocols can reduce a program's performance by requiring expensive, nonlocal operations to invalidate or update shared data in other caches. These operations directly affect the processors referencing the shared data and indirectly slow all processors by increasing contention for the interconnection network and main memory.

Even multiprocessors that are not multis distinguish local and remote memories. On some computers, such as the BBN Butterfly, a portion of each processor's address space is local and can be accessed at low cost. On other machines, such as hypercubes, all memory is local and remote memory can only be referenced through a message to another processor. Programmers on these computers typically cache code and data in a processor's local memory. Some systems for these machines present an illusion of shared memory by caching pages in local memory [9]. Even though these local memories are not managed in hardware, many of the considerations we will discuss are applicable.

In future multiprocessors, the relative cost of coherence protocol operations will be larger than it is at the present time because technological improvements are not reducing communication costs as rapidly as computation times. Users are also demanding increasingly large systems, which require more communication. Properly exploiting all types of locality is critical to using tomorrow's multiprocessors efficiently. Eggers has presented empirical results demonstrating that today's multiprocessor programs frequently misuse the cache, thereby reducing its performance [7].

In the future, perhaps languages and compilers for parallel computers will take the following issues into consideration. In the meantime, the programmer must understand and efficiently use caches in order to take full advantage of multis.

This article restates results by Eggers and other computer architecture researchers in a manner comprehensible to programmers who may know little or nothing about caches. We will also introduce four simple models that help programmers appreciate the implications of multiprocessor caches (summarized in Table I). The Appendix contains a more detailed description of the underlying hardware.

No-Caches Model

The no-caches model assumes that all memory references go to main memory. The advantage of this model is that a programmer does not need to worry about locality, since all memory references are equally expensive. Nonlocal communication can only be reduced by eliminating memory references (i.e., keeping data within a processor's registers or recomputing results). This is the multiprocessor model which many programmers use. It is adequate for the purpose of discussing the functionality of concurrent programs. It fails, however, to capture the need for locality.

Infinite-Word-Caches Model

Our simplest cache model assumes an infinite cache of single memory locations. Once a location is referenced, it remains in a uniprocessor's cache forever. On a multiprocessor, the word disappears from a processor's cache when another processor writes into it.[1]

This model's principal software implication is that programmers should avoid unnecessary interleaving of references by more than one processor to the same memory word (unless all references are reads). To

© 1990 ACM 0001-0782/90/0800-0097 $1.50

The material presented here is based on research supported in part by the National Science Foundation's Presidential Young Investigator and Computer and Computation Research Programs under grants MIPS-8957278 and CCR-8902536, A. T. & T. Bell Laboratories, Cray Research, Digital Equipment Corporation, Texas Instruments, and the graduate school at the University of Wisconsin-Madison.

[1]This model is most accurate for write-invalidate cache-coherence protocols (see Appendix). For the other class of protocols (write-update), this model correctly indicates that nonlocal references are caused by active sharing, but it does not reflect the exact costs of sharing.

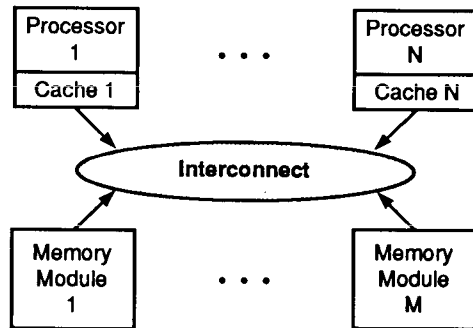

FIGURE 1 A typical cache-coherent, shared-memory multiprocessor (*multi*) has a collection of processors, each with a local cache, connected to main memory modules via an interconnection network. The system's cache-coherence protocol ensures that each processor sees a consistent view of locations in its own cache, just as if all reads and writes went to main memory.

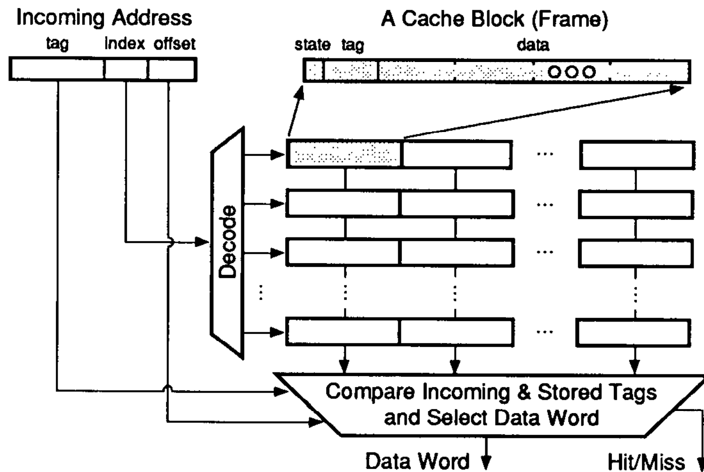

FIGURE 2 This figure illustrates a typical cache. Information is stored in a two-dimensional array of cache block frames. Each frame contains state information (e.g., whether a frame contains valid data), an address tag (the main memory address of data in the frame), and several words of data. The number of words of data in a block is the *block size*; the number of frames in a row is the *associativity*; and the number of frames times the block size is the *cache size*.

On a memory access, the cache partitions the incoming address into three fields: *tag*, *index* and *offset*. The index field selects one row of frames. The tag field is then compared with state and tag fields of each selected frame. For speed, these comparisons occur in parallel. If the block is found (a *hit*), the offset field selects the data word from the appropriate frame. On a miss (not shown), the cache chooses a block to replace, reads the new block from main memory, and then returns the requested data word.

A multiprocessor cache differs from a uniprocessor cache in two ways. First, the cache must respond to network activity in addition to handling normal processor accesses. Second, the state information in each frame expands to include the states of the cache-coherence protocol.

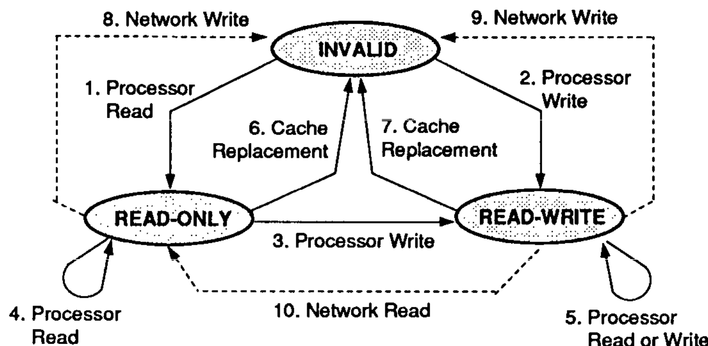

FIGURE 3 A simple write-invalidate cache-coherence protocol has three cache states: INVALID (block not in the cache), READ-ONLY (processor may read from, but not write to the block) and READ-WRITE (reads and writes permitted). State transitions are caused by: processor reads and writes (arcs 1-5); cache block replacements to make room for another block (arcs 6-7); and network (e.g., bus) read and write requests (broken arcs 8-10). Except for arcs 4, 5, 6, and 8, state transitions require network operations: request read-only copy of the block (arc 1), request exclusive copy (2), make copy exclusive (3), update main memory copy (7), send block to requesting processor (9), and update main memory copy and send block to requester (10).

TABLE I. Model Summary

	When a Memory Reference Is Local (Inexpensive)	
Model Name	**Uniprocessor Rule**	**Multiprocessor Rule**
No-Caches	Never.	Never
Infinte-Word-Caches	The processor referenced the location in the past. A *reference* is a read or write.	The processor referenced the location in the past and no other processor wrote into it since this reference.
Infinite-Block-Caches	The processor referenced the location's cache block. A *cache block* is a group of *B* adjacent, aligned words.	The processor referenced the location's cache block and no other processor wrote into a location in the cache block since this reference.
Finite-Block-Caches	The processor referenced the location's cache block *recently*. With a finite cache of *C* words, a reference is recent if it accesses one of the last *C/B* distinct blocks referenced.	The processor referenced the location's cache block recently and no other processor wrote into a location in the cache block since this reference.

appreciate this point, consider the common programming paradigm of maintaining a central queue of tasks and having a process running on each processor remove tasks, execute them, and return new tasks to the queue. Although the arrrangement described is convenient, it ignores locality since a datum may be modified by many tasks executing on different processors. Nonlocal operations will transfer the modified location between the processors' caches. If writes are frequent enough, the traffic generated by these operations will heavily load the memory system and can reduce the whole system's performance. One way to avoid this problem is to maintain a separate task queue for each processor [3][2]. In this case, repeated operations on an object will usually execute on the same processor. If a processor empties its queue, it can remove tasks from another processor's queue.

The interleaving problem can be most severe for variables used for interprocess synchronization, such as locks. A `test-and-set` operation that obtains a lock always modifies a memory location, regardless of whether the lock is free. After a process executes a `test-and-set`, the lock resides exclusively in that processor's cache. Two or more processes contending for a lock aggravate the situation by causing the lock to "ping-pong" between caches, generating large amounts of network traffic and slowing other processors. A simple solution is to test the state of the lock before performing a `test-and-set` instruction [11]. Only when the lock is free, should the more expensive operations be used:

```
repeat
  / * Wait until lock is free before trying
      test-and-set*/
      while (lock ≠ Free) do skip od;
until (test-and-set(lock) = Free);
```

With proper care, this solution, called `test-and-test-and-set`, works well for processors connected through a shared bus [3]. An equivalent technique for synchronization over more general interconnection networks is currently the subject of research [8, 13].

Infinite-Block-Caches Model

Most real caches do not hold individual memory locations. Instead, they hold groups of words surrounding the referenced locations. These word groups form what is called a *block*, and are loaded together when any constituent location is referenced. Blocks of size *B* words are usually *aligned*, meaning that the address of the first word is a multiple of *B*. Typical values for *B* are 4, 8 or 16 words. Cache blocks exploit spatial locality. A program typically uses data in locations near the word it is currently referencing. These nearby words are brought into the cache along with the first referenced location.

These blocks, however, may cause problems when different processors modify adjacent locations. The first write transfers the block to one processor's cache. The second write moves it to the other processor's cache. This sequence is called *false sharing* since no information is transferred [7]. False sharing arises when the data of two processors lie adjacent in memory. For example, in

```
declare integer data [100];
declare lock lock [100];
```

each element of a data vector is protected by a lock in the lock vector. If locks occupy a single memory word and cache blocks contain four words (typical values), a block could hold four different locks, each of which may ping-pong among eight different processors, no more than two of which ever use it. A more effective way to arrange this data is to group related items together and keep unrelated items in separate cache blocks:

```
structure dataNlock {
  integer data;
  lock lock;
  /* Cache blocks are 4 words long */
  integer pad1, pad 2;}
declare dataNlock lockeddata[100];
```

The last two fields (*pad1* and *pad2*) enlarge the structures so each lock-value pair resides in a distinct cache block (assuming that the array *lockeddata* is allocated starting on a four-word boundary).

Finite-Block-Caches Model

One feature not accounted for by the above models is the finite size of real caches, which often hold only 1K to

[2]This has the additional benefit of reducing the bottleneck caused by a single queue.

64K words. A cache of size C words with B-word blocks tends to contain the C/B blocks surrounding the most recent memory references. Finite caches limit locality on both uniprocessors and multiprocessors.

In uniprocessors, limited caches are the principal cause of cache misses. To reduce the number of misses, data should be organized with common access patterns referencing adjacent words; this enables the cache to hold the last C-referenced words. If references are B or more addresses apart, the cache holds only the last C/B-referenced words. In this case, the effective cache size is reduced by a factor of B (which is often 4 to 8). In addition, a programmer should try to reuse words before they are pushed out of the cache. For instance, consider arithmetic operations on vectors. The following two loops compute $A \leftarrow B \times C$ and $E \leftarrow A \times D$, where A, B, C, D, and E, are vectors of length N.

```
for i←1 to N do
  A[i]←B[i] * C[i];
od;
for j←1 to N do
  E[j]←A[j] * D[j];
od;
```

If N is large, when the first loop finishes, the first locations of A may have been flushed from the cache. A better approach is to write these loops as a single loop and use values before they are flushed from the cache:[3]

```
for i←1 to N do
  A[i]←B[i] * C[i];
  E[i]←A[i] * D[i];
od;
```

Optimizing programs to take into account finite caches is less important on multiprocessors than uniprocessors. In many programs; the finite cache size will not be the dominant cause of cache misses. Many misses will be the result of factors discussed previously. Also, reducing cache misses is more complex on a multiprocessor due to interactions with other processors. For example, a change that keeps more items in a cache by packing them tightly may introduce false sharing between processors, degrading performance. Programmers should not optimize multiprocessor programs for finite caches unless the amount of data each processor uses is very large and the changes do not cause harmful interactions with other processors.

[3]On vector machines with caches, programmers (or compilers) may have to compromise vectorizability to attain cache performance. [10]. In this example, however, coalescing the loops does not prevent vectorization.

Conclusion

A program running on a multiprocessor no longer has a single, sequential order of execution. The temporal and spatial locality of a processor is easily disturbed by actions of other processors. Some of these interactions are visible to a programmer, while others are artifacts of hardware. A programmer who understands the basics of multiprocessor caches can reduce the extraneous interference and improve a program's performance.

Here are three rules-of-thumb to consider when writing a parallel program:

1. Try to perform all operations on a datum in the same processor to avoid unnecessary communication.
2. Align data to prevent locations used by different processors from occupying the same cache block.
3. Cluster work and re-use parts of the data quickly, instead of making long passes over all the data.

Programming languages do not currently facilitate this style of programming. A programmer must be aware of the underlying behavior of the multis and write programs that properly exploit shared caches.

Acknowledgments.
We wish to thank Paul Adams, Eric Bach, Renato De Leone, Susan Eggers, Susan Horwitz, Douglas Johnson, Luigi Semenzato, Peter Sweeney, Mary Vernon, and David

Wood for reading and improving drafts of this article.

References

1. Agarwal, A. and Gupta, A. Memory-Reference Characteristics of Multiprocessor Applications under MACH. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Santa Fe, N.M., 1988) pp. 215–226.
2. Agarwal, A., Simoni, R., Horowitz, M., and Hennessy, J. An Evaluation of Discretionary Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture* (Hawaii 1988) pp. 280–289.
3. Anderson, T.E., Lazowska, E.D., and Levy, H.M. The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Berkeley, Calif. 1989) pp. 49–60.
4. Archibald, J. and Baer, J.-L. Cache-Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Trans. Comput. Syst. 4*, 4 (1986) 273–298.
5. Bell, C.G. Multis: A New Class of Multiprocessor Computers. *Science, 228* (1985), 462–466.
6. Eggers, S.J. and Katz, R.H. A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture* (Honolulu, Hawaii 1988), pp. 373–382.
7. Eggers, S.J. and Katz, R.H. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)* (Boston, Mass. 1989) pp. 257–270.
8. Goodman, J.R., Vernon, M.K., and Woest, P.J. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)* (Boston, Mass. April 1989), pp. 64–77.
9. Li, K. and Hudak, P. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst. 7*, 4 (November 1989), 321–359.
10. Liu, B. and Strother, N. Programming in VS Fortran on the IBM 3090 for Maximum Vector Performance. *IEEE Computer, 21*, 6 (June 1988), 65–76.
11. Rudolph, L. and Segall, Z. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture* (Ann Arbor, Mich. 1984) 340–347.
12. Smith, A.J. Cache Memories. *ACM Comput. Surv., 14*, 3 (1982) 473–530.
13. Yew, P.-C., Tzeng, N.-F. and Lawrie, D.H.

Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Trans. Comput., C-36* (1987) 388–395.

CR Categories and Subject Descriptors: B.O. [**Hardware**]: General; B.3 [**Memory Structures**]; B.3.2 [**Design Styles**]: Cache Memories; C.O. [**Computer System Organization**]: General; C.1.2 [**Multiple Data Stream Architecture**]: Parallel Processors; D.O. [**Software**]: General; D.1. [**Programming Techniques**]; D.1.3. [Concurrent Programmers]

General Terms: Performance

Additional Key Words and Phrases: Cache coherence, shared-memory multiprocessors.

About the Authors:

MARK D. HILL is an assistant professor in the Computer Sciences Department at the University of Wisconsin, Madison. His research interests center on computer architecture, with an emphasis on performance considerations and implementation factors in memory systems.

JAMES R. LARUS *is an assistant professor in the* Computer Sciences Department at the University of Wisconsin, Madison. His research interests center on programming languages and compilers for parallel computers.

Authors' Present Address: Computer Sciences Dept., 1210 West Dayton St., Univ. of Wisconsin-Madison, WI 53706, markhill@cs.wisc.edu. and larus@cs.wisc.edu.

APPENDIX

Caches In More Detail

The body of this article explains the software implications of multiprocessor caches. This appendix explains details of how these caches operate for readers who wish to understand the basis of the implications. We first argue that virtual memory and uniprocessor caches are similar, and then discuss how multiprocessing complicates caches.

Hardware caches operate on the same principle as virtual memory. Pages of memory reside on disks, whose access time is much larger than that of physical or main memory. To reduce average access time, a virtual memory system (operating system software, often with microcode or hardware support) keeps copies of the most recently referenced pages in main memory. When memory is updated, a disk page becomes out-of-date or *stale*. Users never access stale data, because the virtual memory system directs references to the memory copy when one exists and always updates the disk page before the memory copy is replaced.

Uniprocessor caches function like virtual memory, except that the faster level of storage is the cache (usually fast, static RAM), while the slower level is main memory (usually large, dynamic RAM). Cache pages are called blocks or lines, and cache management is handled totally by hardware. Figure 2 shows the structure of a typical cache.

A multiprocessor with per-processor caches is more complex, because data also becomes stale when another processor updates it. Consider the case in which processor 1 has updated a cache block, but not main memory; processor 2 does not have a copy of this block; and then processor 2 references the block. Some mechanism must ensure that processor 2 receives the updated copy from processor 1's cache, not stale copy from main memory. Otherwise, a programmer's model of a shared, cache-less memory is compromised. This mechanism is called a *cache-coherence protocol*.

For computers with more than four processors, the first commercial systems with cache-coherence protocols connected processors and main memory through a single, shared bus. A bus simplifies the coherence protocol by providing inexpensive, atomic broadcasts. Multiprocessors with a bus exploit this capability by having all processors (actually the processors' cache controllers) monitor bus transactions. When a transaction affects a location in a processor's cache, the controller updates the cache, places data on the bus, or both.

Bus-based cache-coherence protocols can be classified as *write-invalidate* or *write-update* [4, 6]. Write-invalidate protocols guarantee that there exist either: (1) no cached copies of a block, (2) one or more read-only copies, or (3) one read-write copy. Bus transactions maintain this invariant. The protocols are called write-invalidate because a processor wishing to write a block invalidates all read-only copies. Figure 3 illustrates a simple write-invalidate protocol. Most of these protocols' overhead is due to invalidate operations and the subsequent cache misses incurred by other processors when they re-reference invalidated blocks.

Write-update cache-coherence protocols allow multiple read-write copies, but require that each update be broadcast, preventing the existence of stale copies. These protocols usually contain a mechanism allowing a writing processor to determine that no other cache copies exist, so subsequent writes need not be broadcast. Write-update protocols increase the cost of all writes to shared blocks, but eliminate the re-reference misses of write-invalidate protocols.

The obvious bandwidth limitations of a single, shared bus have led researchers and hardware designers to investigate cache-coherence protocols on more general interconnection networks. On these networks, broadcasts are expensive and often non-atomic. Many broadcasts can be avoided by adding a level of indirection. Instead of issuing a broadcast request to all processors, a protocol can look at a known location, called a *directory entry*, to get pointer(s) to a block's cached location(s). The protocol can then communicate directly with the processors that have copies of a location. Agarwal *et al.* extend a write-invalidate protocol to this type of computer [2]. Contrary to wide-spread belief, access to directory entries does not introduce a centralized bottleneck since entries for different blocks can be in different places. Write-update protocols appear less amenable to multiprocessors with general interconnection networks, because updates are difficult to propagate atomically and efficiently. ■