CACHE CONSIDERATIONS FOR PROGRAMMERS OF MULTIPROCESSORS

by

Mark D. Hill
and
James R. Larus

Computer Sciences Technical Report #891

November 1989

Cache Considerations for Programmers of Multiprocessors

Mark D. Hill* and James R. Larus
Computer Science Department
1210 West Dayton St.
University of Wisconsin–Madison
Madison, WI 53706

November 8, 1989

Abstract

Although caches in computers are invisible to programmers, they significantly affect programs' performance. This is particularly true for multiprocessors. This paper presents results from recent computer architecture research about the performance of parallel programs and discusses their implications for programmers who may know little or nothing about caches.

1 Introduction

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* is a high-speed buffer adjacent to a processor that holds copies of memory locations likely to be used soon [11]. 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.

*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, Digital Equipment Corporation, Texas Instruments, and the graduate school at the University of Wisconsin–Madison.

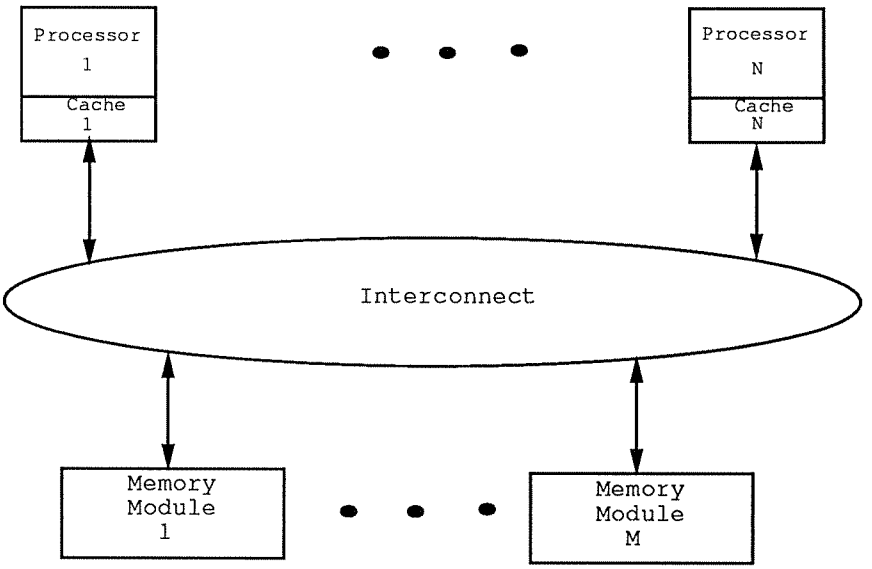

Figure 1: A typical cache-coherent, shared-memory multiprocessor (*multi* for short) has a collection of processors, each with a local cache, connected to main memory modules via an interconnection network. The system's cache coherence protocol ensures that each processor sees a consistent view of locations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

Parallel computing introduces a new type of locality, called *processor locality*, in which contemporaneous references to a memory word come from a single processor, not many different ones [1]. Many multiprocessors devote substantial resources to provide a large cache for each processor and then hide this feature so programmers can write correct programs without reasoning about these caches. Although programmers find it possible (and 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 paper describes the behavior of these hidden caches and presents some guidelines for a programmer who wishes to use them more effectively.

We are most interested in cache-coherent, shared-memory multiprocessors, *multis* for short [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, non-local 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, some 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. Even though these caches are not managed in hardware, many of the considerations discussed below apply.

In future multiprocessors, the relative cost of coherence protocol operations will be larger than today because technological improvements are not reducing communication costs as rapidly as computation times. Users are also demanding larger and larger systems, which require more communication. Properly exploiting all types of locality is critical to using tomorrow's multiprocessors efficiently. Eggers presented empirical results demonstrating that misuse of a multi's cache is common in today's programs and it reduces their performance [6].

This paper restates results from this and other computer architecture research in a manner comprehensible to programmers who may know little or nothing about caches. The next four sections introduce four simple models that help programmers appreciate the implications of multiprocessor caches. Appendix A contains a more detailed description of the underlying hardware for readers interested in knowing more. Table 1 summarizes the models: *No-Caches*, *Infinite-Word-Caches*, *Infinite-Block-Caches* and *Finite-Block-Caches*.

2 No-Caches Model

The no-caches model assumes that all memory references go to main memory. This model's advantage is that a programmer need not worry about locality, as all memory references are equally (very) expensive. Non-local communication can only be reduced by eliminating memory references, for example by keeping data within a processor's registers or by re-computing results.

When a Memory Reference is Local (Inexpensive)		
Model Name	Uniprocessor Rule	Multiprocessor Rule
No-Caches	Never.	Never.
Infinite-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.

This is many programmers' model of a multiprocessor and it suffices to discuss the functionality of concurrent programs. It fails, however, to capture the need for locality.

3 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. To see 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 convenient, this paradigm ignores locality since a datum may be modified by many tasks that execute on different processors. Non-local operations will transfer the modified location between the processors' caches. If writes are frequent enough, the traffic

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

generated by these operations heavily loads 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, thereby 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 [10]. Only when the lock is free, invoke the more expensive operations:

```
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 [9, 12].

4 Infinite-Block-Caches Model

Real caches do not hold individual memory locations. Instead, they hold groups of words surrounding the referenced locations. These words, called a *block*, 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

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

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 [8]. False sharing arises when two processes' data lies 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 better 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, pad2;}

declare dataNlock lockeddata[100];
```

5 Finite-Block-Caches Model

One feature not accounted for by the models above is the finite size of real caches, which often hold only 1K to 100K words. Finite caches limit locality on both uniprocessors and multiprocessors. A cache of size C words with B-word blocks tends to contain the C/B blocks surrounding the most-recent memory references. This fact has two implications for a programmer. First, it is advisable to organize data so common access patterns reference adjacent words, allowing the cache to hold the last C referenced words. For, if references are B or more addresses apart, the

cache only holds 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. Second, try to reuse words before they are pushed out of the cache. For example, 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 early locations of A may have been flushed from the cache, so a better approach is to write these loops as a single loop and use values before they are flushed from the cache:

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

6 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.

Below 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 so locations used by different processors do not land in the same cache block.

3. Cluster work and re-use parts of the data quickly, rather than making long passes over all the data.

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

7 Acknowledgements

We wish to thank Paul Adams, Eric Bach, Renato De Leone, Susan Eggers, Susan Horwitz, Douglas Johnson, Luigi Semenzato, Peter Sweeney, and Mary Vernon for reading and improving drafts of this paper.

Appendix

A Caches In More Detail

The body of this paper explained 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.

Recall 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, not stale main memory, copy. 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 processor's cache controller) listen to bus transactions and, when necessary, update their caches, place data on the bus, or both.

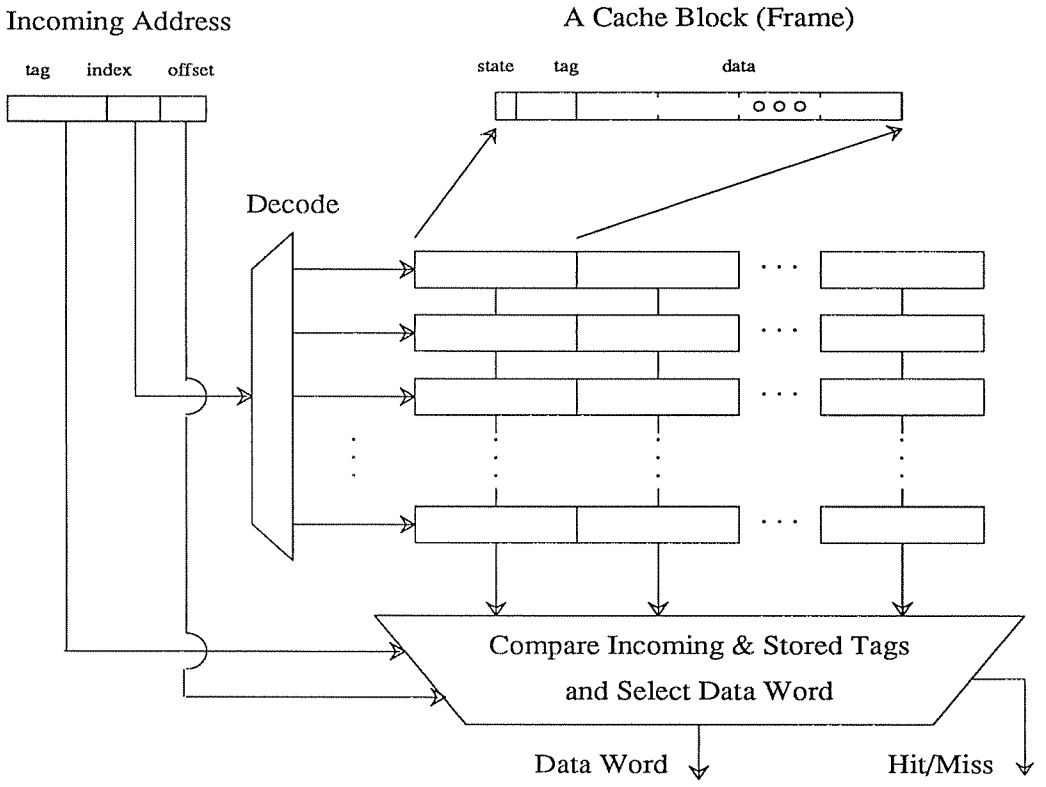

Figure 2: This figure illustrates a typical cache. Information is stored in a two dimensional array of cache block frames. Each frame contains state information (e.g., whether a frame contains valid data), an address tag (the main memory address of data in the frame), and several words of data. The number 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 coherency protocol (described below).

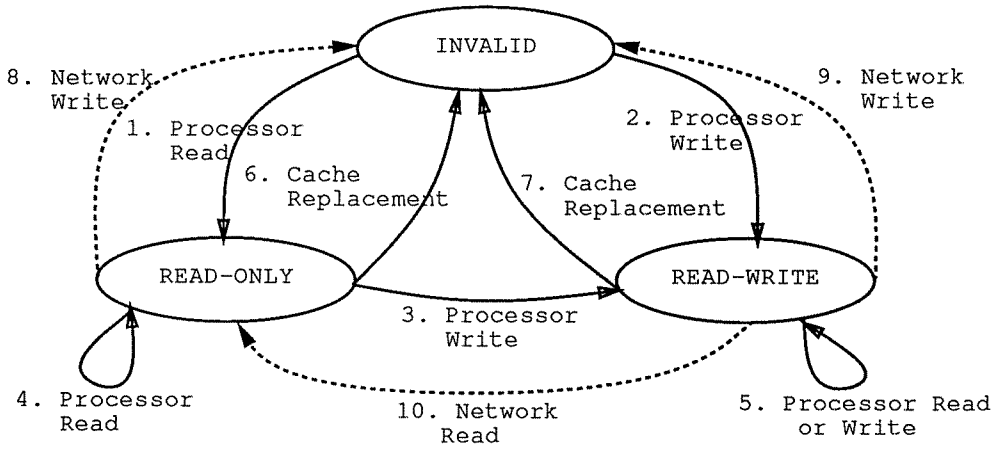

Figure 3: A simple write-invalidate cache-coherence protocol has three cache states: INVALID (block not in the cache), READ-ONLY (processor may read from, but not write to the block) and READ-WRITE (reads and writes permitted). State transitions are caused by: processor reads and writes (arcs 1-5), cache 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).

Bus-based cache-coherence protocols can be classified as *write-invalidate* or *write-update* [7]. Write-invalidate protocols guarantee that there are 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 so no stale copies exist. Usually, these protocols contain a mechanism that allows 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 lead 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.

References

[1] Anant Agarwal and Anoop Gupta. Memory-Reference Characteristics of Multiprocessor Applications under MACH. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 215–226, 1988.

[2] Anant Agarwal, Richard Simoni, Mark Horowitz, and John Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, 1988.

[3] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 49–60, 1989.

[4] J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

[5] C. Gordon Bell. Multis: A New Class of Multiprocessor Computers. *Science*, 228:462–466, 1985.

[6] Susan J. Eggers. Simplicity Versus Accuracy in a Model of Cache Coherency Overhead. Submitted to IEEE Transactions on Computers, 1989.

[7] Susan J. Eggers and Randy H. Katz. 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*, pages 373–382, 1988.

[8] Susan J. Eggers and Randy H. Katz. 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)*, pages 257–270, 1989.

[9] James R. Goodman, Mary K. Vernon, and Philip J. Woest. 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)*, pages 64–77, April 1989.

[10] Larry Rudolph and Zary Segall. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 340–347, 1984.

[11] Alan J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[12] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, 1987.