ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/220772017

An Evaluation of Directory Schemes for Cache Coherence.

Conference Paper *in* ACM SIGARCH Computer Architecture News · January 1998

DOI: 10.1109/ISCA.1988.5238 · Source: DBLP

CITATIONS
241

READS
67

4 authors, including:

Anant Agarwal
Massachusetts Institute of Technology
342 PUBLICATIONS **13,326** CITATIONS
SEE PROFILE

John L. Hennessy
Stanford University
291 PUBLICATIONS **24,008** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Computer Architecture: A Quantitative Approach: Sixth Edition View project

All content following this page was uploaded by John L. Hennessy on 10 September 2014.

The user has requested enhancement of the downloaded file.

An Evaluation of Directory Schemes for Cache Coherence

Anant Agarwal,[*] Richard Simoni, John Hennessy, and Mark Horowitz
Computer Systems Laboratory
Stanford University, CA 94305

Abstract

The problem of cache coherence in shared-memory multiprocessors has been addressed using two basic approaches: directory schemes and snoopy cache schemes. Directory schemes have been given less attention in the past several years, while snoopy cache methods have become extremely popular. Directory schemes for cache coherence are potentially attractive in large multiprocessor systems that are beyond the scaling limits of the snoopy cache schemes. Slight modifications to directory schemes can make them competitive in performance with snoopy cache schemes for small multiprocessors. Trace driven simulation, using data collected from several real multiprocessor applications, is used to compare the performance of standard directory schemes, modifications to these schemes, and snoopy cache protocols.

1 Introduction

In the past several years, shared-memory multiprocessors have gained wide-spread attention due to the simplicity of the shared-memory parallel programming model. However, allowing the processors to share memory complicates the design of the memory hierarchy. The most prominent example of this is the *cache coherency* or *cache consistency* problem, which is introduced if the system includes caches for each processor. A system of caches is said to be *coherent* if all copies of a main memory location in multiple caches remain consistent when the contents of that memory location are modified [1]. A *cache coherency protocol* is the mechanism by which the coherency of the caches is maintained. Maintaining coherency entails taking special action when one processor writes to a block of data that exists in other caches. The data in the other caches, which is now stale, must be either invalidated or updated with the new value, depending on the protocol. Similarly, if a read miss occurs on a shared data item and memory has not been updated with the most recent value (as would happen in a copy-back cache), that most recent value must be found and supplied to the cache that missed. These two actions are the essence of all cache coherency protocols. The protocols differ primarily in how they determine whether the block is shared, how they find out where block copies reside, and how they invalidate or update copies.

Most of the consistency schemes that have been or are being implemented in multiprocessors are called *snoopy cache* protocols [2, 3, 4, 5, 6, 7] because each cache in the system must watch all coherency transactions to determine when consistency-related actions should take place for shared data. Snoopy cache schemes store the state of each block of cached data in the cache directories – the information about the state of the cached data is distributed.

Another class of coherency protocols is *directory-based* [8, 9, 10, 11]. Directory-based protocols keep a separate directory associated with main memory that stores the state of each block of main memory. Each entry in this centralized directory may contain several fields depending on the protocol, for example, a dirty bit, a bit indicating whether or not the block is cached, pointers to the caches that contain the block, etc.

How do snoopy cache protocols work? A typical scheme enforces consistency by allowing multiple readers but only one writer. The state associated with a block's cached copy denotes whether the block is, for example, (i) invalid, (ii), valid (possibly shared), or (iii) dirty (exclusive copy). When a cache miss occurs, the address is *broadcast* on the shared bus. If another cache has the block in state dirty, the state is changed to valid and the block is supplied to the requesting cache. In addition, for write misses all copies of the block are invalidated. Similarly, on a write hit to a clean block, the address is broadcast and each cache must invalidate its copy. In general, all cache transactions that may require a data transfer or state change in other caches must be broadcast over the bus.

Snoopy cache schemes are popular because small-scale multiprocessors can live within the bandwidth constraints imposed by a single, shared bus to memory. This shared bus makes the implementation of the broadcast actions straightforward. However, snoopy cache schemes will not scale beyond the range of the number of processors that can be accommodated on a bus (probably no more than 20). Attempts to scale them by replacing the bus with a higher bandwidth communication network will not be successful since the consistency protocol relies on low-latency broadcasts to maintain coherency. For this reason, shared-memory multiprocessors with large numbers of processors, such as the RP3 [12], do not provide cache coherency support in hardware.

These snoopy cache schemes also interfere with the processor-cache connection. Because the caches of *all* processors are examined on each coherency transaction, interference between the processor and its cache is unavoidable. This interference can be reduced by duplicating the tags and snooping on the duplicate tags. However, the processor must write both sets of tags and thus arbitration is required on the duplicate tags. This impacts the cache write time which may slow down the overall cycle time, especially in a high performance machine. Attempts to reduce the bus traffic generated by cache coherency requests in a snoopy cache scheme results in fairly complex protocols. These may impact either the cache access time or the coherency trans-

[*]Anant Agarwal is currently with the Laboratory for Computer Science (NE43-418), M.I.T, Cambridge, MA 02139.

In *Proceedings of the 15th Annual International Symposiumn on Computer Architecture (ISCA)*,
Honolulu, Hawaii, May 30 - June 2, 1988, pages 280-289.

action time.

In this paper we propose that directory-based schemes are better suited to building large-scale, cache-coherent multiprocessors, where a single bus is unsuitable for a communication mechanism. This paper is a first step in evaluating directory schemes using traces from real multiprocess applications. Although we do not have sufficient data to demonstrate quantitatively that the directory schemes are effective in a large-scale multiprocessor, we do discuss how these directory schemes can be scaled and we demonstrate that their performance in a small-scale multiprocessor is acceptable.

We use trace-driven simulation, with traces obtained from real multiprocessor applications, to evaluate a basic directory-based coherency protocol that uses bus broadcasts and verify that its performance approaches that of snoopy cache schemes. We then obviate broadcasts by including a valid bit per cache in each directory entry, allowing sequential invalidation of multiple cached copies. Performance is not significantly degraded by this modification, and in most cases (over 85% of writes to previously-clean blocks) no more than one sequential invalidation request is necessary. Unfortunately, the need for a valid bit per cache restricts the ability to add on to an existing multiprocessor without modifying parts of the existing system. This motivates a scheme that can perform up to some small number of sequential invalidates to handle the most frequent case, and that resorts to some form of "limited broadcast" otherwise.

The paper first reviews previous directory schemes and discusses how they overcome the limitations created by snoopy cache schemes. It also proposes a general classification of these techniques, and identifies a few that seem most interesting for performance and implementation reasons. Section 3 outlines the schemes that we evaluate. We describe our evaluation method and the characteristics of our multiprocessor address traces in Section 4. Section 5 evaluates basic directory and snoopy cache schemes and discusses their performance. Section 6 then extends the discussion to include more scalable directory protocols, and Section 7 concludes the paper.

2 Directory Schemes for Cache Consistency

The major problems that snoopy cache schemes possess are limited scalability and interference with the processor-cache write path. How do directory schemes address these problems? The major advantage directory schemes have over snooping protocols is that the location of the caches that have a copy of a shared data item are known. This means that a broadcast is not required to find all the shared copies. Instead, individual messages can be sent to the caches with copies when an invalidate occurs. Since these messages are directed (i.e., not broadcast), they can be easily sent over any arbitrary interconnection network, as opposed to just a bus. The absence of broadcasts eliminates the major limitation on scaling cache coherent multiprocessors to a large number of processors.

Because we no longer need to examine every cache for a copy of the data, the duplicate tags can be eliminated. Instead, we store pointers in main memory to the caches where the data is known to reside and invalidate their copies. The protocols are also simpler than the distributed snoopy algorithms because of the centralization of the information about each datum.

Several directory-based consistency schemes have been proposed in the literature. Tang's method [8] allows clean blocks to exist in many caches, but disallows dirty blocks from residing in more than one cache (most snoopy cache coherency schemes use the same policy). In this scheme, each cache maintains a dirty bit for each of its blocks, and the central directory kept at memory contains a copy of all the tags and dirty bits in each cache. On a read miss, the central directory is checked to see if the block is dirty in another cache. If so, consistency is maintained by copying the dirty block back to memory before supplying the data; if the directory indicates the data is not dirty in another cache, then it supplies the data from memory. The directory is then updated to indicate that the requesting cache now has a clean copy of the data. The central directory is also checked on a write miss. In this case, if the block is dirty in another cache then the block is first flushed from that cache back to memory before supplying the data; if the block is clean in other caches then it is invalidated in those caches (i.e., removed from the caches). The data is then supplied to the requesting cache and the directory modified to show that the cache has a dirty copy of the block. On a write hit, the cache's dirty bit is checked. If the block is already dirty, there is no need to check the central directory, so the write can proceed immediately. If the block is clean, then the cache notifies the central directory, which must invalidate the block in all of the other caches where it resides.

Censier and Feautrier [9] proposed a similar consistency mechanism that performs the same actions as the Tang scheme but organizes the central directory differently. Tang duplicates each of the individual cache directories as his main directory. To find out which caches contain a block, Tang's scheme must search each of these duplicate directories. In the Censier and Feautrier central directory, a dirty bit and a number of valid (or "present") bits equal to the number of caches are associated with each block in main memory. This organization provides the same information as the duplicate cache directory method but allows this information to be accessed directly using the address supplied to the central directory by the requesting cache. Each valid bit is set if the corresponding cache contains a valid copy of the block. Since a dirty block can only exist in at most one cache, no more than one of a block's valid bits may be set if the dirty bit is set.

Yen and Fu suggest a small refinement [11] to the Censier and Feautrier consistency technique. The central directory is unchanged, but in addition to the valid and dirty bits, a flag called the *single bit* is associated with each block in the caches. A cache block's single bit is set if and only if that cache is the only one in the system that contains the block. This saves having to complete a directory access before writing to a clean block that is not cached elsewhere. The major drawback of this scheme is that extra bus bandwidth is consumed to keep the single bits updated in all the caches. Thus, the scheme saves central directory accesses, but does not reduce the number of bus accesses versus the Censier and Feautrier protocol.

Archibald and Baer present a directory-based consistency mechanism [10] with a different organization for the central directory that reduces the amount of storage space in the directory, and also makes it easier to add more caches to the system. The directory saves only two bits with each block in main memory. These bits encode one of four possible states: *block not cached*, *block clean in exactly one cache*, *block clean in an unknown number of caches*, and *block dirty in exactly one cache*. The directory therefore contains no information to indicate which caches contain a block; the scheme relies on broadcasts to perform invalidates and write-back requests. The *block clean in exactly one cache* state obviates the need for a broadcast when writing to a clean block that is not contained in any other caches.

Two clear differences are present among these directory schemes: the number of processor indices contained in the directories and the presence of a broadcast bit. We can thus classify the schemes as Dir_iX, where i is the number of indices kept in the directory and X is either B or NB for Broadcast or No Broadcast. In a no-broadcast scheme the number of processors that have copies of a datum must always be less than or equal to i, the number of indices kept in the directory. If the scheme allows broadcast then the numbers of processors can be larger and when it is (indicated by a bit in the directory) a broadcast is used to invalidate the cached data. The one case that does not make sense is Dir_0NB, since there is no way to obtain exclusive access.

In this terminology, the Tang scheme is classified as Dir_nNB, the Censier and Feautrier scheme is Dir_nNB also, and the Baer and Archibald scheme is Dir_0B. Our evaluation concentrates on a couple of key points in the design space: Dir_1NB and Dir_0B. We will also present results for Dir_nNB.

There are two potential difficulties that prevent scalability of the directory schemes. First, if the scheme always or frequently requires broadcast, then it will do no better than the snoopy schemes. Variations in the directory schemes (e.g., increasing the value of i in a Dir_iB scheme) decrease the frequency of broadcast. We must also examine the dynamic numbers of caches that contain a shared datum to evaluate the actual frequency of occurrence. Second, the access to the directory is a potential bottleneck. However, we will show that the directory is not much more of a bottleneck than main memory, and the bandwidth to both can be increased by having a distributed memory hierarchy rather than centralized. That is, memory is distributed together with individual processors. In addition to certain advantages in providing scalable bandwidth to the memories from the local processor, the organization distributes the directory, associating it with the individual memory modules.

3 Schemes Evaluated

We will evaluate two directory schemes (called Dir_1NB and Dir_0B), and two snoopy cache schemes (Write-Through-With-Invalidate and Dragon) for comparison purposes. These particular snoopy cache techniques were selected because they represent two extremes of performance and complexity. The two directory schemes are also extremes in the number of simultaneous cached copies allowed. The following is a description of these four protocols.

The most restrictive of the four schemes is Dir_1NB in that a given block is allowed to reside in no more than one cache at a time; therefore, there can be no data inconsistency across caches. The directory entry for each block consists of a pointer to the cache that contains the block. On a cache miss, the directory is accessed to find out which cache contains the block, that cache is notified to invalidate the block and write it back to memory if dirty, and the data is then supplied to the requesting cache. Dir_1NB is included in the evaluation because it is perhaps the simplest directory-based consistency scheme and is easily scaled to support a large number of processors.

The Dir_0B is the Archibald and Baer scheme [10] outlined in the previous section. Like many consistency protocols, a clean block may reside in many caches, while a dirty block may exist in exactly one cache. Invalidations are accomplished with broadcasts; a similar scheme that uses sequential invalidates in place of broadcasts (Dir_nNB) will later be shown to have nearly the same performance. For the initial evaluation, broadcasts are used in both the directory and snooping schemes because it results in a simpler cost model and allows a fair comparison of the two.

Write-Through-With-Invalidate (WTI) is a simple snoopy cache protocol that relies on a write-through (as opposed to copy-back) cache policy and is used in several commercial multiprocessors. All writes to cache blocks are transmitted to main memory. Other caches snooping on the bus check to see if they have the block that is being written; if so, they invalidate that block in their own cache. When a different processor accesses the block, a cache miss will occur and the current data will be read from memory. Like Dir_0B, multiple cached copies of clean blocks can exist simultaneously. Because of the high level of bus traffic caused by the write-through strategy, WTI is generally considered to be one of the lowest-performance snooping cache consistency protocols.

While the three previous schemes are all invalidation protocols, Dragon is an update protocol, i.e., it maintains consistency by updating stale cached data with the new value rather than by invalidating the stale data [13]. The cache keeps state with each block to indicate whether or not each block is shared; all writes to shared blocks must be broadcast on the bus so that the other copies can be updated. Dragon uses a special "shared" line to determine whether a block is currently being shared or not. Each cache snoops on the bus and pulls the shared line whenever it sees an address for which it has a cached copy of the data. Dragon is often considered to have the best performance among snoopy cache schemes.

4 Evaluation Methodology

Simulation using multiprocessor address traces is our method of evaluation. Most previous studies that evaluated directory schemes used analytical models [14, 9] and those that used simulation had to make rough assumptions about the characteristics of shared memory references [10]. Because the performance of cache coherence schemes is very sensitive to the shared-memory reference patterns, both of these previous methods have the drawback that the results are highly dependent on the assumptions made. Trace-driven simulation has the drawback that the same trace is used to evaluate all consistency protocols, while in reality the reference pattern would be different for each of the schemes due to their timing differences. But the traces represent at least one possible run of a real program, and can accurately distinguish the performance of various schemes for that run.

This paper deals with the inherent cost of sharing in multiprocessors and the memory traffic required to maintain cache consistency. We therefore exclude the misses caused by the first reference to a block in the trace because these occur in a uniprocessor infinite cache as well. The additional overhead due to multiprocessing now consists of (i) the extra misses that occur due to fetching the block into multiple caches and (ii) the cache consistency-related operations. Our results represent exactly this overhead.

We wish to isolate and measure only the traffic incurred in maintaining a coherent shared memory system in a multiprocessor. To this end our simulations use infinite caches to eliminate the traffic caused by interference in finite caches. The performance of an infinite cache is also a good approximation to that of a very large cache, where the miss rate is essentially the cost of first-time fetches. Moreover, the performance of a system with smaller caches can be estimated to first order by adding the costs due to the finite cache size. Typical cache miss rates are reported in [15, 16].

4.1 Performance Measures

To determine the absolute performance of a multiprocessor system using total processor utilizations, a simulation must be carried out for every hardware model desired. A problem with this approach is that the sharing characteristics may change because the simulation model is different from the hardware used for gathering data.

We would like a metric for performance that is not tied to any particular processor or interconnection network architecture. We use the communication cost per memory reference as our basic metric. This cost is simply the average number of cycles that the bus (or network) is busy during a data transfer from a cache to another cache, cache to directory, and from cache to or from main memory. We refer to this metric simply as bus cycles per memory reference. This metric abstracts away details of how the directories are implemented, either as centralized or distributed. It also requires no assumptions about the relative speeds of local and non-local memories, local and non-local buses, or processor and the bus.

Since the snoopy cache schemes require a bus-based architecture, we often talk of a bus in our directory models. However, the directory schemes we discuss are general enough to work in any network architecture. While the bus cycles metric allows us to compare the relative merits of various cache consistency schemes, it cannot indicate accurately the absolute performance of a multiprocessor. However, in lightly loaded systems, multiprocessor performance could still be approximated to first order from the number of bus cycles used per memory reference.

The bus cycles per reference for a given cache consistency scheme are computed as follows. First we measure event frequencies for various schemes by simulating multiple infinite caches, where events are different types of memory references. The simulator reads a reference from a trace and takes a set of actions depending on the type of the reference, the state of the referenced block, and the given cache consistency protocol.

The event frequencies are now weighted by their respective costs in bus cycles to give the aggregate number of bus cycles used per reference. For example, a cache miss event might require 5 bus cycles of communication cost (1 cycle to send the address, and 4 cycles to get 4 words of data back). If the rate of cache misses is, say, 1%, then the bus cycles used up by cache misses per reference is 0.05. In like manner, the costs due to other events are added to get the aggregate cost per reference. Since the choice of the hardware model (i.e., cost per event) is independent of the event frequencies, we need just one simulation run per protocol to compute the event frequencies, and we can then vary costs for different hardware models.

Details of traces used in simulations are given in Section 4.4. The block size used throughout this paper is 4 words (16 bytes). In all the schemes we assume that instructions do not cause any cache consistency related traffic. In addition, we do not include the bus traffic caused by instruction misses in our performance estimations.

4.2 Event Frequencies

The event types of interest in a particular scheme are those that may result in a bus transaction. All the schemes require the frequency of read and write misses (*read-miss* or *rm* and *write-miss* or *wm*. Depending on the scheme some other events rates are also needed:

- The Dragon events include the fraction of references to blocks that are clean or dirty in another cache on a read or write miss (*rm-blk-cln, rm-blk-drty, wm-blk-cln,* and *wm-blk-drty*). The clean and dirty numbers indicate when a block is supplied by another cache as opposed to from main memory. In addition, we need the frequency of write updates to blocks present in multiple caches on a write hit (*wh-distrib*).
- The write-through scheme requires the frequency of writes (*write*) because all writes are transmitted to main memory.
- In the Dir_1NB scheme, we need the fractions of read and write references that miss in the cache, but are present in a dirty or clean state in another cache (*rm-blk-cln, rm-blk-drty, wm-blk-cln,* and *wm-blk-drty*). These events indicate when invalidation requests must be sent to another cache and when dirty blocks have to be written back to main memory.
- In the Dir_0B scheme, in addition to the four events for the Dir_1NB scheme, we need the proportion of write hits to a clean block (*wh-blk-cln*). This event represents queries to the directory to check whether the block resides in any other cache and has to be invalidated. We also measure the distribution of the number of caches the block resides in during a possible invalidation situation to determine the impact of various invalidations methods. The various invalidation methods include full broadcast, limited broadcast, and sequential invalidation messages to each cache.

4.3 Bus Models

The bus cycle costs for the various events depend on the sophistication of the bus and main memory. The examples given in this paper use the bus timing depicted in Table 1. From this basic bus model, and some assumptions about the sophistication of the bus, we can estimate the cost in bus cycles for each of the events that cause bus traffic. Because the costs can differ depending on the type of bus or interconnection network used, we will use two bus types of widely diverse complexity to give an idea of how the schemes will perform over a range of bus and memory organizations. On

Table 1: Timing for fundamental bus operations.

Bus Operation	Bus Cycles
Send address	1
Transfer 1 data word	1
Invalidate	1
Wait for Directory	2
Wait for Memory	2
Wait for Cache	1

Table 2: Summary of bus cycle costs.

Access Type	Pipelined Bus	Non-Pipelined Bus
mem access	5	7
cache access	5	6
write back	4	4
invalidate	1	1
wt or wup	1	2
dir access	1	3

the sophisticated end of the spectrum, we use a pipelined bus model that has separate data and address paths. At the other end we use a non-pipelined bus that has to multiplex the address and data on the same bus lines. The data transfer width of both buses is assumed to be one word (32 bits).

For the pipelined bus with separate lines for address and data, memory or non-local cache accesses cost 5 cycles (1 cycle to send the address and 4 cycles to get the data). The bus is not held during the access. Write-backs cost 4 cycles: the first cycle sends the address and the first data word; the remaining 3 words are sent in the next three cycles. When the data is transferred to memory during a write-back, the requesting cache also receives it. The bus cycles used for data transfer are then counted under the write-back category. A write-through to memory or a write update to another cache is 1 cycle. A directory check uses 1 cycle to send the address, and invalidates are also 1 cycle.

In the non-pipelined bus model, the bus has to be held during the memory or non-local cache access. Here a memory access costs 7 cycles, 1 cycle to send the address, 2 cycles to wait for the memory access, and 4 cycles to get the data. An access from another cache is 6 cycles, and takes a cycle less than the memory access because the cache access wait is only one cycle. Write-backs still cost 4 cycles; the waiting for memory is counted under the memory access category, and the bus need not be held while the write into memory is taking place. As in the pipelined bus, the data is also received by the requesting cache on a write-back. A write-through or a write update to another cache is 2 cycles, 1 cycle to send the address and 1 cycle to send the data word. A directory check is 3 cycles, 1 cycle to send the address and 2 cycles to access the directory. When possible the directory access is overlapped with memory access. Invalidations cost 1 cycle. These costs for the pipelined and non-pipelined bus models are summarized in Table 2.

In the non-pipelined bus, once the address and the data have been sent to memory or to another cache on a write (or write-back) operation we assume that the bus need not be held while data is being written into memory. This is a simplifying assumption and is usually true if memory is interleaved. We also assume that broadcast invalidates, like a single invalidate, take 1 cycle. We do not attempt to model the impact of broadcast invalidate on the bus cycle time.

Table 3: Summary of trace characteristics. All numbers are in thousands.

Trace	Refs	Instr	DRd	DWrt	User	Sys
POPS	3142	1624	1257	261	2817	325
THOR	3222	1456	1398	368	2727	495
PERO	3508	1834	1266	409	3242	266

4.4 Multiprocessor Trace Data

The traces used for simulation are obtained using a multiprocessor extension of the ATUM address tracing scheme [17]. The multiprocessor used for tracing was a VAX 8350 with four processors. An address trace contains interleaved address streams of the four processors. CPU numbers and process identifiers of the active processes are also included in the trace so that any address in the trace can be identified as coming from a given CPU and given process. A current limitation of ATUM traces is that only four-CPU traces can be obtained. We are currently developing a multiprocessor simulator that builds on top of the VAX T-bit mechanism and can provide accurate simulated traces of a much larger number of processors.

The traces show some amount of sharing between processors that is induced solely by process migration. The characteristics of migration-induced sharing is significantly different from sharing present in the application processes [18]. We would like to exclude this form of sharing from our study since a large multiprocessor would probably try to minimize process migration. Therefore, for this study, we consider sharing between processes (as opposed to sharing between processors), which means that a block is considered shared only if it is accessed by more than one process. Because the time sequence of the references in the trace is strictly maintained, the temporal ordering of various synchronization activities in the trace, such as getting or releasing a synchronization lock, is still retained. As a check on this model, we collected all our statistics based on both process sharing and processor sharing and found that the numbers were not significantly different. The similarity is due to the few instances of process migration in our traces.

We currently use three traces for this study. The traces are of parallel applications running under the MACH operating system [19]. Table 3 describes the characteristics of the traces used for this study. POPS [20] is a parallel implementation of OPS5, which is a rule-based programming language. THOR is a parallel implementation of a logic simulator done by Larry Soule at Stanford University. PERO is a parallel VLSI router written by Jonathan Rose at Stanford. All traces include operating system activity, which comprises roughly 10% of the traces.

The traces show a larger-than-usual read-to-write reference ratio due to spins on locks in POPS and THOR. The spins correspond to the first test in a *test-and-test-&-set* synchronization primitive. These appear as reads of a data word. Roughly one-third of all the reads correspond to reads due to spinning on a lock. We will look at how the number of spins on a lock affect the performance of cache consistency schemes in Section 5.2. The ratio of reads to writes in PERO is also high, but this reference behavior is a result of the algorithm used in the program.

5 Evaluation of Directory-Based and Snoopy-Cache Protocols

The first step in evaluating the four consistency schemes is to measure the frequency of each type of reference. Table 4 gives a breakdown of the various types of references that take place in the four schemes and their relative frequencies, averaged across the three traces. All numbers in this table are shown as a percentage of the total number of references.

Table 4: Event frequencies. The numbers are shown as a percentage of all references. The fractions in each subcategory add up.

Event	Schemes			
Type	Dir_1NB	WTI	Dir_0B	Dragn
instr	49.72	49.72	49.72	49.72
read	39.82	39.82	39.82	39.82
rd-hit	34.32	38.88	38.88	39.20
rd-miss(rm)	5.18	0.62	0.62	0.30
rm-blk-cln	4.78	-	0.23	0.14
rm-blk-drty	0.40	-	0.40	0.17
rm-first-ref	0.32	0.32	0.32	0.32
nwrite	10.46	10.46	10.46	10.46
wrt-hit(wh)	10.19	10.25	10.25	10.36
wh-blk-cln	-	-	0.41	-
wh-blk-drty	-	-	9.84	-
wh-distrib	-	-	-	1.74
wh-local	-	-	-	8.62
wrt-miss(wm)	0.17	0.12	0.11	0.02
wm-blk-cln	0.08	-	0.02	0.01
wm-blk-drty	0.09	-	0.09	0.01
wm-first-ref	0.08	0.08	0.08	0.08

LEGEND	
instr	Instructions
read	Reads
rd-hit	Read hits
rd-miss(rm)	Read misses
rm-blk-cln	Read miss, blk clean in another cache
rm-blk-drty	Read miss, blk dirty in another cache
rm-first-ref	Read miss, first reference to the blk
write	Writes
wrt-hit(wh)	Write hits
wh-blk-cln	Write hit, blk clean in the same cache
wh-blk-drty	Write hit, blk dirty in the same cache
wh-distrib	Write hit, block also in another cache
wh-local	Write hit, blk not in another cache
wrt-miss(wm)	Write miss
wm-blk-cln	Write miss, blk clean in another cache
wm-blk-drty	Write miss, blk dirty in another cache
wm-first-ref	Write miss, first reference to the blk

We can make several useful observations about the cache behavior as well as the data sharing behavior of the traces from these event counts. The most obvious feature of the numbers for the Dir_1NB consistency scheme is the high rate of data read misses (5.18% of all references), indicating a high penalty for allowing a block to reside in no more than one cache at a time. The Dir_1NB numbers also show a low rate of data write misses (0.17% of all references), which implies that most data writes occur on blocks which have first been brought into the cache via read misses.[1] Furthermore, it is usually the case that no other process accesses those blocks between the read immediately preceding a write and the write itself, since this situation would result in a write miss for the Dir_1NB scheme. The Dir_0B consistency technique, on the other hand, shows a much smaller rate of read misses (0.62%) than Dir_1NB, illustrating that most of the misses incurred in Dir_1NB were caused by read sharing among multiple processes.

[1]Accesses to lock variables – a successful test followed by a test and set in the test-and-test-&-set primitive – are an example.

The fact that the reference rates for the WTI method match those of Dir_0B brings up an interesting point. A cache consistency protocol can be thought of as being made up of two parts: a specification of the state changes of the data in the caches (e.g., when is data brought into the cache, invalidated) and the protocol which is used to accomplish that specification (e.g., write-through with bus snooping, centralized directory). The frequency with which each of the events listed in Table 4 occurs depends only on the state change specification, not on the method used to implement it. Since Dir_0B and WTI both rely on the same basic data state-change model of allowing multiple cached copies of clean blocks but only a single copy of dirty blocks, their event frequencies are identical. (However, they do differ in that Dir_0B allows main memory blocks to become stale with respect to cache blocks. This distinction and the difference in cost associated with some events accounts for their disparity in performance.) This basic state-change model is also found in some other consistency schemes [7] and the event frequencies for Dir_0B and WTI are valid for these as well.

The Dragon consistency mechanism differs from the others in Table 4 because it is an update protocol rather than an invalidation protocol. For this reason the miss rates are very small in an infinite cache; once a block is loaded into a cache, it remains there forever. The most important events for Dragon are not cache misses, but rather write hits that cause a bus transaction. The numbers in the table indicate that roughly one-sixth of all writes require a bus broadcast to perform a write update.

Viewing the event frequencies in absolute terms (rather than in relation to the frequencies in other schemes) can provide some insight into the amount of overhead generated by enforcing cache consistency in a multiprocessor. One simple metric of this overhead (for an invalidation protocol) is the increase in the cache miss rate due to the invalidations required to ensure consistency. Since they were generated using infinite caches, the miss rates in Table 4 are an upper bound on the amount by which the miss rate of a finite-sized cache will increase.[2] From Table 4 we can compute the component of the miss rate due to invalidations caused by cache coherency. Because there are no invalidations in the Dragon scheme, its miss rate is the native miss rate for these traces. From Table 4, the data component of the native miss rate is 0.72%. Therefore, the difference between the Dir_0B data miss rate and the native data miss rate is $1.13 - 0.72 = 0.41\%$ which is the miss rate component due to cache coherency. Consistency-related misses therefore comprise $0.41/1.13 = 36\%$ of the total miss rate.

In invalidation schemes like Dir_0B, a write to a previously-clean block must invalidate that block in all other caches that have a copy. This is the case for two events in Table 4: *wm-blk-cln* and *wh-blk-cln*. Figure 1 shows the

[2]The coherency-related misses will be fewer in a finite-sized cache because some of the blocks that would be invalidated to enforce consistency in an infinite cache have already been purged in a finite cache due to cache interference.

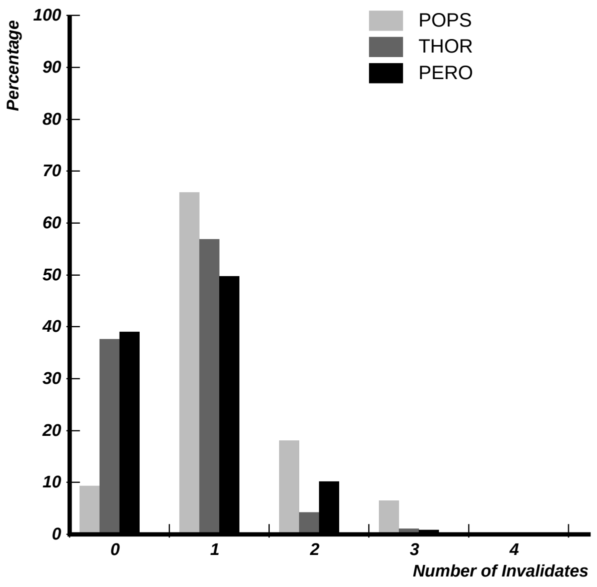

Figure 1: Number of caches in which a block must be invalidated on a write to a previously-clean block.

histogram of the number of other caches that contain a previously-clean block when it is written (i.e., when one of the above two events occurs). This number is equal to the number of caches in which a clean block must be invalidated when it is written. The figure shows that on average, over 85% of the writes to previously-clean blocks cause invalidations in no more than one cache.[3] This points out the inefficiency in using a bus broadcast to accomplish the invalidation operation, and suggests some possible enhancements to directory-based consistency schemes which will be discussed shortly.

Figure 2 shows the average number of bus cycles per reference, calculated as described in Section 4. The two endpoints of each bar represent the performance of the pipelined and non-pipelined buses. The performance of Dir_0B approaches that of the Dragon scheme for this metric. Not surprisingly, Dir_1NB and WTI are much worse than Dir_0B and Dragon. As observed in [1], Dragon shows the best performance because the cost of a write update is assumed to be much lower than the cost of an invalidation and a subsequent miss. Figure 3 shows the average number of bus cycles per reference for the individual traces. The numbers for the POPS and THOR traces are similar, while those for PERO are much smaller. The chief reason is that the fraction of references to shared blocks in PERO is much smaller than in POPS and THOR. Another observation is that the relative performance of the four schemes does not depend strongly on the sophistication of the bus. For the remainder of the paper we will focus on the pipelined bus for brevity.

Table 5 shows the breakdown by operation of the aver-

[3]The number of times that invalidations occur in no more than one cache, computed as a fraction of all references that may require invalidations, is even larger. Such references, in addition to writes that occur to previously-clean blocks, include read/write misses to blocks dirty in another cache (*rm-blk-drty*, *wm-blk-drty*), which require exactly one invalidation.

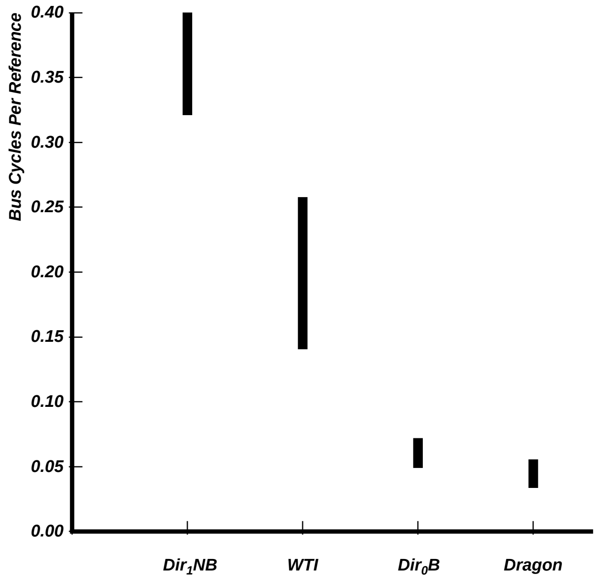

Figure 2: Range of bus cycle requirements (average). The low value of each bar corresponds to the pipelined bus and high value to the non-pipelined bus.

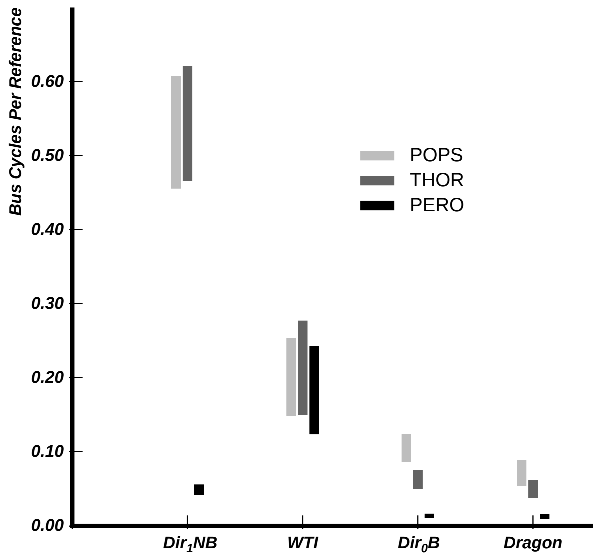

Figure 3: Range of bus cycle requirements for the individual traces. The low value of each bar corresponds to the pipelined bus and high value to the non-pipelined bus.

age number of bus cycles per reference. Dir_0B is shown to use close to 50% more bus cycles than the Dragon scheme (0.0491 versus 0.0336). The performance of Dir_0B in a real system is closer to Dragon than this metric indicates because the fixed costs of references which use the bus impact Dragon more severely, as pointed out in Section 5.1.

As an interesting aside, the Dir_0B event frequencies can be used to estimate the performance of the Berkeley Ownership protocol [7] by modifying the Dir_0B cost model. The cost models are different because Dir_0B has to probe the directory to find out whether it needs to do an invalidate, while the Berkeley scheme gets this information from the state of the block in the cache.[4] The cost model for the Berkeley scheme is thus derived from the Dir_0B scheme by trivially setting the directory access cost to 0 bus cycles. With this model, the number of bus cycles consumed by an average reference in the Berkeley scheme is 0.0409, placing it roughly midway between the Dir_0B and Dragon schemes.

Table 5: Breakdown of bus cycles for the pipelined bus. The category "wt or wup" stands for write-through in the WTI scheme and write update in the Dragon scheme. Note that directory accesses can always be overlapped with memory accesses in Dir_1NB.

Access Type	Schemes			
	Dir_1NB	WTI	Dir_0B	Dragon
mem access	0.2479	0.0369	0.0173	0.0160
write back	0.0196	-	0.0196	-
invalidate	0.0535	-	0.0081	-
wt or wup	-	0.1037	-	0.0176
dir access	-	-	0.0041	-
cumulative	0.3210	0.1406	0.0491	0.0336

The data in Table 5 is shown graphically in Figure 4. The figure depicts the breakdown of the bus cycles as a fraction of the total number of bus cycles used by each scheme, highlighting the relative importance of various events. In Dir_1NB, for instance, the high miss rate on clean blocks makes the number of bus cycles spent on invalidations and write-backs small compared to the number of memory accesses. Not surprisingly, most of the bus cycles consumed in WTI are due to the write-through cache policy. The Dragon scheme splits its bus cycles evenly between loading up each cache with data and using the bus on write hits to keep that data consistent.

In Dir_0B, the number of cycles used for directory accesses that cannot be overlapped with memory accesses is small relative to the total number of cycles. This result diminishes previous concerns that the directory itself could be a major performance bottleneck. In fact, the required directory bandwidth is only slightly higher than the bandwidth to memory. Techniques used to increase available memory bandwidth, such as distributing memory with the individual processors, can be applied to the directory as well. The fraction of cycles spent on invalidations is low, which implies that increasing this cost by a small factor will cause a relatively small increase in the total number of bus cycles used by Dir_0B. This result indicates that invalidating data in caches sequentially (rather than using a bus broadcast) may be viable without severely degrading performance. This case will be evaluated further in Section 6.

[4] The Berkeley scheme, in addition, uses a different state for a dirty block that becomes shared to enable the cache to supply a block rather than memory. This optimization does not impact our performance metric in the pipelined bus.

The data in Table 5 can be used to determine the system performance in a shared-bus environment. The number of bus cycles consumed by a reference in the best scheme with a sophisticated bus is about 0.03 on average. In other words, a processor will use a bus cycle every 30 references, or a bus cycle every 15 instructions since on average each instruction in the traces makes one data reference. (We assume instruction misses do not cause bus transactions.) A 10-MIPS processor will therefore require a bus cycle every 1500ns, and a bus with a cycle time of 100ns will only yield a maximum performance of 15 effective processors. This limit is an optimistic upper bound because we have not included the bandwidth requirements of instruction misses, the effects of finite data caches, or the effects of bus contention.

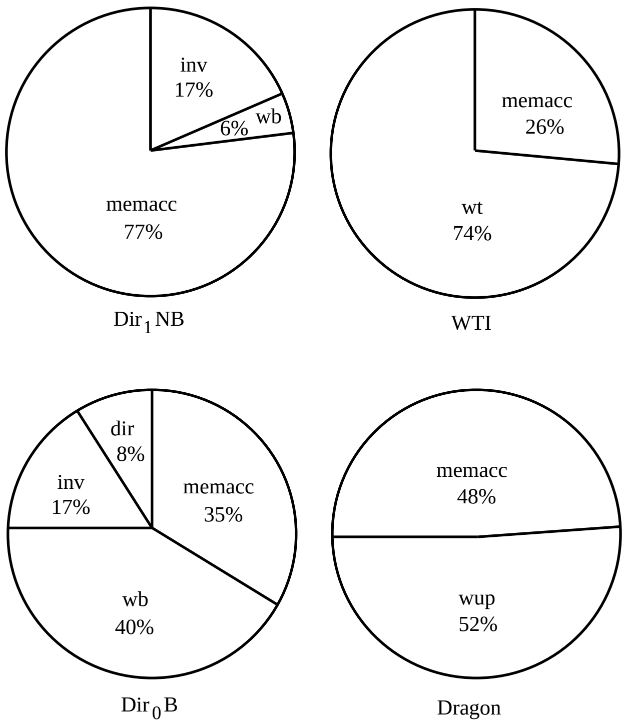

Figure 4: Bus cycle breakdown in the various schemes as a fraction of the total number of bus cycles used in the scheme. The code *dir* is directory access, *inv* is invalidate, *wb* is write-back, *memacc* is memory access, *wt* is write through, and *wup* is write update.

5.1 System Performance

Total system performance cannot be determined from the bus cycles metric alone. A better metric for this purpose is average memory access time as seen by each processor, but this metric requires many assumptions about the implementation of the memory hierarchy. Regardless of the memory system details, it is clear that the additional waiting time beyond the number of bus cycles for a reference as seen by the processor will be at least one bus cycle time. This additional "bus cycle" is used for initial cache access, propagation delay through the bus controller, and bus arbitration. Figure 5 shows the average number of bus cycles per bus transaction for each of the schemes. Because the average Dragon cost is smaller than Dir_0B, the performance of the Dragon scheme will be more sensitive to changing that cost by a constant value. Consequently, for the metric of average memory access time as seen by the processor, we would expect Dragon to show less of an advantage than with the bus cycles metric.

Even using the bus cycles metric we can get an idea of the effect of adding a small constant number of bus cycles to the cost models. If q bus cycles are added to the cost of each bus transaction, the performance for Dragon is given by $0.0336 + 0.0206q$ and the performance for Dir_0B is given by $0.0491 + 0.0114q$ bus cycles per reference. For example, with $q = 1$ Dir_0B needs only 12% more bus cycles than Dragon, as compared with 46% in Figure 2.

5.2 Impact of Spin Locks on Cache Consistency Performance

Spin lock reads severely degrade the performance of the Dir_1NB scheme as measured by our bus cycles metric. The number of bus cycles in Dir_1NB is over a factor of six greater than the number used by Dir_0B. As mentioned earlier one-third of the reads in POPS and THOR are due to spins on a lock. Because two processes often spin on the same lock, locks frequently bounce back and forth between two caches in the Dir_1NB scheme. To verify this phenomenon, we ran a set of experiments excluding all the tests on locks in the three traces. As expected Dir_0B gave the same performance as before, while the performance of Dir_1NB improved significantly (from 0.32 to 0.12 bus cycles per reference).

The impact of spin locking on the performance of the Dir_1NB scheme is also interesting in another light. Software cache consistency schemes that flush a critical section from the cache after each use will behave like the Dir_1NB scheme. For reasonable performance, these schemes must take special care in handling locks.

6 Directory Scheme Alternatives for Scalability

The need to perform full broadcasts limits the potential to scale a multiprocessor to a large number of processors. To obviate full broadcasts, pointers to all caches containing a block can be maintained in the directory (Dir_nNB [9]). In this scheme, sequential invalidations are sent to each of the caches denoted by the pointers instead of using a full broadcast. We evaluated this scheme assuming that each invalidation consumes one bus cycle. The number of bus cycles per reference for a pipelined bus increases from 0.0491 in the full broadcast case (Dir_0B) to 0.0499 in the sequential invalidate case (Dir_nNB). The performance degradation is small because often no more than one invalidation is necessary.[5]

Although the sequential invalidation scheme has comparable performance to the broadcast scheme, the directory size increases in proportion to the number of processors. The next scheme that we discuss capitalizes on the observation that a single invalidation request in the most common case. The directory maintains exactly one pointer and a broadcast

[5]Note that our data was obtained from a machine with only four processors. We are trying to obtain traces for a much larger number of processes and hope to extend our results shortly.

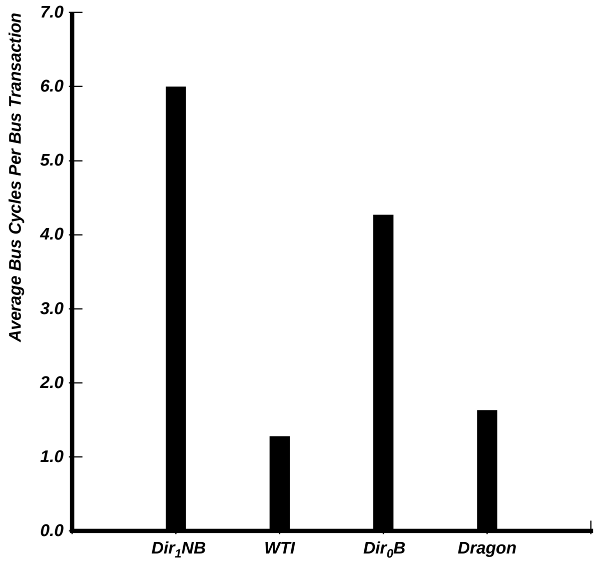

Figure 5: Average bus cycles per bus transaction in various schemes.

bit per block (Dir_1B). If more than one cache has a block the broadcast bit is set. When the directory is queried, a single invalidation request is issued if the broadcast bit is clear; otherwise, the invalidation must be broadcast. While it is hard to quantize the exact effects of broadcasts, the following simple model can help indicate the performance of such a scheme. Suppose that a single invalidation takes one bus cycle as usual, and that a broadcast uses b cycles. With this simple model, this directory scheme requires $0.0485{+}0.0006b$ cycles per memory reference. This scheme can be extended to use i pointers ($i > 1$) and a broadcast bit (Dir_iB). The broadcast requirement can be eliminated if the number of allowed copies of a block is restricted to i as in the scheme denoted Dir_iNB. This scheme trades off a slightly increased miss rate for avoiding broadcasts altogether.

We can also use *limited broadcasts* if the caches where block copies exist are known. The number of bits in the main memory directory can be reduced by storing a simple code representing a set of caches, which is a superset of all caches with a copy of the block. For example, consider storing a word with d digits where each digit takes on one of three values: $0, 1$, and *both*. If each digit in the word is either a 0 or a 1, then the word is the index to exactly one cache in the system. If any digit is coded *both*, then the word denotes caches whose indices may either be a 0 or a 1 in that digit, but match the rest of the word. If i digits are coded *both*, then 2^i caches are denoted. In like manner, we can code a set of caches that includes all block copies. Each digit can be coded in 2 bits, thus requiring $2log(n)$ bits in a system with n caches.

As the above examples show, a class of directory schemes exist that can trade off a small amount of performance for scalability and ease of implementation. An accurate evaluation of the tradeoffs will require traces from a much larger number of processors.

7 Conclusions

This paper shows that directory-based cache consistency schemes are an interesting approach for providing shared memory in a large-scale multiprocessor. The directory structure removes the major limitation of snoopy-cache schemes – the reliance on broadcasts – while providing similar efficiency in handling shared references. The bandwidth requirement to the directory, long considered a potential bottleneck, is shown to be not much more severe than the memory bandwidth need. The basic bandwidth limitation to the memory and the directory can be mitigated by distributing them on the processor boards. This technique allows the bandwidth to both the memory and the directory to scale with the number of processors.

We evaluated the performance of directory schemes in a small-scale multiprocessor environment using trace driven simulation. The performance of the directory protocols is reasonably competitive to the snoopy cache schemes. In addition the simulations show that most blocks that are written into are present in only a small number of other caches, which makes broadcast invalidates inefficient. This result suggests that a directory structure that stores with each block only a small number of pointers to caches containing the block is sufficient. If this data holds for large-scale multiprocessors, directories will provide an efficient method of implementing shared memory.

8 Acknowledgements

Our thanks to Roberto Bisiani and the Speech Group at CMU for letting us use their VAX 8350 to obtain traces. Dick Sites made multiprocessor ATUM possible, and Digital Equipment Corporation lent the ATUM microcode to us. Jonathan Rose helped with the PERO program and Larry Soule with THOR. The research reported in this paper was funded by DARPA contract MDA903-83-C-0335. Richard Simoni is partially supported by a National Science Foundation Graduate Fellowship.

References

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

[2] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual Symposium on Computer Architecture*, pages 124–131, June 1983.

[3] Charles P. Thacker and Lawrence C. Stewart. Firefly: a Multiprocessor Workstation. In *Proceedings of ASPLOS II*, pages 164–172, October 1987.

[4] Frank S. J. Tightly Coupled Multiprocessor System Speeds Up Memory Access Times. Electronics, 57, 1, January 1984.

[5] Mark S. Papamarcos and Janak H. patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 348–354, June 1985.

[6] L. Rudolph and Z. Segall. Dynamic Decentralized Cache Consistency Schemes for MIMD Parallel Processors. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 340–347, June 1985.

[7] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a Cache Consistency Protocol. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 276–283, June 1985.

[8] C. K. Tang. Cache Design in the Tightly Coupled Multiprocessor System. In *AFIPS Conference Proceedings, National Computer Conference, NY, NY*, pages 749–753, June 1976.

[9] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, c-27(12):1112–1118, December 1978.

[10] James Archibald and Jean-Loup Baer. An Economical Solution to the Cache Coherence Problem. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 355–362, June 1985.

[11] Wei C. Yen, David W. L. Yen, and King-Sun Fu. Data Coherence Problem in a Multicache System. *IEEE Transactions on Computers*, c-34(1):56–65, January 1985.

[12] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings ICPP*, pages 764–771, August 1985.

[13] E. McCreight. The Dragon Computer System: An Early Overview. Technical report, Xerox Corp., September 1984.

[14] Michel Dubios and Faye A. Briggs. Effects of Cache Coherence in Multiprocessors. In *Proceedings of the 9th International Symposium on Computer Architecture*, pages 299–308, May 1982.

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

[16] Anant Agarwal. *Analysis of Cache Performance for Operating Systems and Multiprogramming.* PhD thesis, Stanford University, May 1987. Available as Computer Systems Laboratory TR-87-332.

[17] Richard L. Sites and Anant Agarwal. Multiprocessor Cache Analysis using ATUM. In *Proceedings of the 15th International Symposium on Computer Architecture*, June 1988.

[18] Anant Agarwal and Anoop Gupta. Memory-Reference Characteristics of Multiprocessor Applications under MACH. In *Proceedings of SIGMETRICS 1988*, May 1988.

[19] Robert Baron, Richard Rashid, Ellen Siegel, Avadis Tevanian, and Michael Young. MACH-1: An Operating System Environment for Large-Scale Multiprocessor Applications. *IEEE Software*, July 1985.

[20] Anoop Gupta, Charles Forgy, and Robert Wedig. Parallel Architectures and Algorithms for Rule-Based Systems. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, June 1986.

View publication stats