Cache Coherence in a Multiport Memory Environment

Scott E. Crawford and Ronald F. DeMara
Department of Electrical and Computer Engineering
University of Central Florida
Orlando, Florida USA 32816-2450
Tel: 407 823-5916 E-Mail: rfd@engr.ucf.edu

Abstract:

The effects of various cache coherence strategies are analyzed for a multiported shared-memory multiprocessor. Analytical models for Concurrent-Read-Exclusive-Write access (CREW) and Concurrent-Read-Concurrent-Write access (CRCW) are developed including shared-not-cacheable, snooping-bus, snooping-bus with cache-to-cache transfers, and directory protocols. The performance of each protocol is shown as the hit rate, main-memory-to-cache-memory cycle-time ratio, fraction of shared data, read percentage, and number of partitions are varied. Overall, results indicate that a snooping-bus with cache-to-cache transfer scheme provides consistently fast access times over a wide range of execution parameters. However, nearly equivalent performance can be obtained with simpler directory-based schemes. The implications of these results on increasing port complexity and memory usage are discussed.

Keywords: Multiprocessor Cache Coherence, Shared-Memory Multiprocessors, Multiport Memory, Snooping Bus, Directory-Based Coherence Protocol.

1 Introduction

In recent years, multiprocessor systems have become an extremely important and widespread topic in the computer architecture community. Additional processors within a system have drawn increased attention to the bandwidth mismatch between CPUs and main memory. Effective caching techniques are imperative in multiprocessor systems, since adding a cache memory for each processor greatly mitigates the delay in accessing main memory.

However, with the appearance of multiple caches, the problem of *cache coherence* surfaced. If multiple caches are allowed to hold simultaneous copies of a given memory location, some mechanism must exist to ensure that all copies remain consistent when one is modified.

Many early multiprocessor systems simply consisted of multiple processors sharing a common bus to main memory. Cache coherence protocols were developed for this architecture to alleviate the consistency problem.

As technology advanced, the techniques that were feasible for a common-bus architecture were not scalable to large numbers of processors and non-bus architectures. Other schemes were developed for use in distributed systems, and these have remained current for several years.

Recently, however, multiported memories have been developed to allow multiple simultaneous access to memory. These memories provide several hardware interfaces, each identical to that of a single static RAM [17], establishing a new approach in the design of multiprocessor and parallel systems. Since multiport memories are recent advancements, it is not yet clear how current caching protocols may be applied.

2 Previous Work

Although cache coherency has been a popular topic for many years, there seems to be little, if anything, written specifically to address the problem of cache coherence in a multiport memory system.

Methods for ensuring cache consistency generally fall into one of four major classes:

- *snooping* techniques for shared-bus systems
- directory-based protocols
- static (shared writable data is non-cacheable)
- software-based protocols

The relative merits of each technique depend, in part, on the specific architecture to which it will be applied. In addition, the first two schemes are regarded as hardware techniques and are implemented by the cache controller with no software support. The last two schemes rely on the compiler (or programmer) to provide additional information to maintain consistency.

2.1 Static (Shared Writable Data Non-Cacheable)

Perhaps the simplest cache coherency strategy is a static

coherence check [10][15]. At compile time, any shared writable data is tagged as non-cachable. In this manner, any block that can be modified by more than one processor will exist only in main memory, and there will be no multiple copies to keep consistent. Any read or write of shared data bypasses the cache and accesses main memory directly. This technique will be referred to as No-Cache throughout this paper.

Although this protocol is the easiest to implement, it can be a poor performer, since up to 40% of all references can be to shared writable data [15] stored in slow main memory.

An alternative method is to have separate caches for shared and private data, thus increasing the speed of memory accesses to common information. However, in a multiport memory environment, a common cache would defeat the purpose of having multiple access channels, and would therefore be inappropriate in such a system.

2.2 Snooping Protocols

A great deal has been written about the cache coherence problem in shared-bus multiprocessor systems [2][3][12]. Since several processors share a common memory via a single bus, cache coherence strategies in these cases generally rely on so-called *snoopy* protocols, where the bus is monitored by each cache in the system. Each bus transaction is monitored to determine if the cache has a copy of the block being accessed. If so, appropriate action would be taken to maintain consistency between copies of the data.

Unfortunately, single bus systems suffer from severe contention problems as the number of processors increases, since all processors must vie for the common bus. Since multiport memories can be used (in part) to overcome the problems of bus/memory contention, the schemes based on common-bus protocols would seem fvunsuitable. However, this is not necessarily the case, as illustrated below.

In recent years, variations on the bus-based theme have surfaced as collections of buses arranged in trees, cubes, or interleaves [15][4]. As long as the number of buses remains small, the number of snooping interfaces remains low, and the cost is not exorbitant.

Similar methods could be applied to multiport memories, as each port of the memory can be viewed as a separate bus. Multiple busses may be snooped, and as long as the number of ports is limited, the cost of snooping interfaces does not become excessive. However, as the number of ports increases, the number of snooping interfaces increases $O(n^2)$, as each cache controller must snoop every other port. The associated hardware cost could easily become prohibitive.

To mitigate the high cost of snooping multiple buses, an alternative architecture may be used which requires less hardware. Each cache controller determines if other caches require information regarding its memory accesses. If no other caches must be notified, the memory access proceeds normally. If, however, other caches need to be apprised, the controller performs the access via a separate bus known as the snooping bus. All cache controllers vie for use of this single snooping bus, thereby limiting the number of required interfaces to one per cache controller. Although this scheme creates contention problems for the common bus, hardware costs for this scheme only grow $O(n)$, rather than $O(n^2)$.

2.2.1 Synapse Protocol

One example of a multiple-bus snooping scheme is implemented in the Synapse N+1 multiprocessor [3]. It uses a snoopy bus-based protocol for its two system buses and 28 processors. In this scheme, cache block frames are defined as existing in one of the states listed in Table 1, below.

State	Contents of Block Frame
INVALID	invalid data
VALID	valid, unmodified, possibly shared data
DIRTY	the only valid copy in the system, modified

Table 1 Synapse Cache Frame States

Cache coherency is maintained by obeying the following policies:

- READ HIT: Data is forwarded to the processor with no action by the protocol required.
- READ MISS: If another cache has a DIRTY copy, the requesting cache is denied the memory block. The cache that owns the DIRTY copy writes the data to main memory and invalidates its local copy. When the original requesting cache re-requests the data, it will be provided from main memory.
- WRITE HIT: If the block's local state is DIRTY, no action by the protocol is required. If the block's local state is VALID, the state is changed to DIRTY on the write, and the block is written to main memory via the snooping bus to ensure all other caches invalidate their data.
- WRITE MISS: If another cache has a DIRTY copy, the sequence of events described in the

READ MISS section will occur. If other caches have VALID copies, all caches invalidate their copies and the block loaded into the requesting cache is tagged DIRTY.

2.2.2 Firefly Protocol

Another snoopy protocol is used in the Firefly, a multiprocessor workstation by Digital Equipment Corporation [3]. This method's significant difference from other strategies is that direct cache-to-cache transfers are used to fetch blocks after a cache miss. This policy can dramatically reduce overall access time, since main memory need only be accessed if no cache holds a copy of the data in question. Defined states for blocks in cache are enumerated in Table 2.

State	Contents of Block Frame
VALID-EXCLUSIVE	the only cached copy, unmodified
DIRTY	the only cached copy, modified
SHARED	unmodified data that other caches may also hold

Table 2 Firefly Cache Frame States

Note that there is no INVALID state: Firefly maintains coherence by updating every cached copy of data on each write.

Coherence is ensured by obeying the following protocol [3]:

- READ HIT: Data is forwarded to the processor with no action by the protocol required.
- READ MISS: If another cache has a copy of the block, it is sent to the requesting cache. All caches then set their state to SHARED. If the block was DIRTY, the owner must write the block to main memory simultaneously. If no other cache holds a copy, it is supplied by main memory, and loaded as VALID-EXCLUSIVE.
- WRITE HIT: If the block is DIRTY or VALID-EXCLUSIVE, the write proceeds immediately, and the state is changed to DIRTY if required. If the block is SHARED, the snooping bus must be obtained, and the word written to main memory. All caches observe this write to memory, and update the values of their local copies.
- WRITE MISS: As with a read miss, the block will be supplied by another cache if possible, or main memory if no cache holds a copy. If the block came from memory, it is loaded DIRTY, and the write proceeds without delay. If it came from another cache, the block is loaded SHARED, so the requesting cache must write the word to main memory via the snooping bus, as described for a write hit.

2.3 Directory-based Protocols

In a directory based scheme [5] [10] [1] [15], physical memory is divided into fixed-size blocks. A directory of blocks is maintained with each block's entry consisting of 1 bit per cache (representing whether the block is present in that cache), a modified bit, and a lock bit. Using the notation of [5], blocks exist in one of the states listed in Table 3.

State	Memory Block Status
ABSENT	no copies exist in any caches
PRESENT	unmodified copies exist in one or more caches
PRESENTM	exactly one cache holds a copy, modified
LOCKED	operation is in progress on the block

Table 3 Block States for Directory Protocol

Similarly, each cache maintains a directory for its block frames with two bits per entry: a valid bit and a lock bit. Together, these bits provide state information for block frames in each cache. Table 4 enumerates these states.

State	Contents of Block Frame
INVALID	invalid data
VALID	valid, unmodified data
VALIDM	valid, modified data

Table 4 Directory Cache Frame States

Cache coherence is maintained by ensuring that when a block is modified, all other copies are invalidated. This allows multiple copies of a block to exist in different caches, as long as the copies are read-only. A write to any copy causes all others to be invalidated.

Directory-based schemes have been shown to be more

scalable than snooping-based protocols [19]. As multiport memory systems are developed with greater numbers of processors, these techniques might exhibit distinct advantages over bus-based algorithms.

2.4 Software-based Protocols

Software-based protocols [6][16][14] rely on a complex and sophisticated compiler to insert cache control instructions at compile-time to maintain coherence. Such schemes could force the cache to flush its contents to main memory whenever a critical section of code was exited. More complex strategies could monitor cache accesses and force cache flushes on writes only. Such protocols, although perhaps feasible, require either laborious programming or elaborate compilers, and are beyond the scope of this paper.

3 Analytical Models

To compare various cache coherence strategies, analytical models were developed for each technique. These models project the average time per memory access for each strategy, based on assumptions justified in the following sections. Since memory access times vary dramatically among computer systems, the calculational results are not intended to be interpreted as strict numerical data. Rather, by making assumptions consistent throughout all models, *trend analysis* as parameters are varied, and *relational analysis* between the techniques are valid regardless of the specific numerical results.

3.1 Basic Assumptions

Hardware assumptions for the memory system include a four-ported main memory with caches and processors connected to three of the ports. The last port is used as the snooping bus for the Synapse and Firefly protocols, and is unused for the other techniques. For the directory scheme, a separate four-ported fast memory is reserved for storage of the directory. As above, three ports serve as access paths for the caches, and the fourth is unused. This architecture is shown in Figure 1.

All techniques are assumed to use *read forwarding*: on a read miss, the requested word is forwarded through the cache to the processor as it is read from main memory. In this manner, there is no additional time penalty to transfer data from cache to the processor after the block has been loaded into the cache.

Multiported memories are typically designed with one of two access protocols: *concurrent read exclusive write (CREW)*, or *concurrent read concurrent write (CRCW)*. Both conventions allow multiple simultaneous reads of

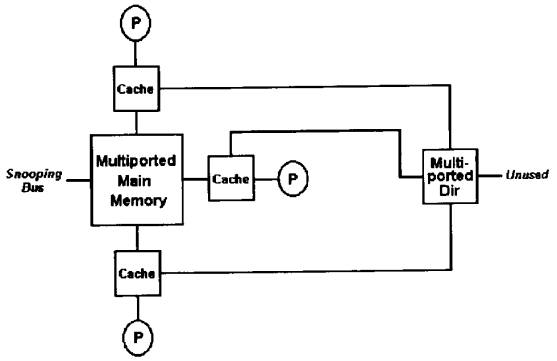

Figure 1 Memory Architecture

memory, but the first only allows one port to write to memory at a time — a restriction not enforced for CRCW.

Since it is impractical to design a memory with fully concurrent writes, an approximation is often established: Memory is divided into several partitions, and concurrent writes are allowed as long as they access different partitions. This closely models the CRCW philosophy. All strategies were analyzed using both CREW and CRCW simulations.

To ensure that these metrics evaluate only the coherence strategies themselves without being biased by other factors, several assumptions are required. An infinite cache size was assumed so interference due to page replacement in finite caches is removed. Main memory was also assigned infinite size to ensure that results are not distorted by page faults requiring slow disk accesses. A separate supervisor operating system is chosen to inhibit process migration. This will prevent page faults due to a process changing processors (and therefore caches) during its lifetime. These suppositions are reasonable, since the performance of an infinite cache is a good approximation of a very large cache, and smaller cache performance can be readily estimated off the larger [1].

Lastly, assume that the application to be executed is homogeneous, and can be evenly distributed between all processors and caches. This implies each cache achieves identical hit rates, utilization of shared data, read/write percentages, and address distribution of main memory accesses.

3.2 No-Cache

Since shared data is maintained exclusively in main memory, the time required to access any such data is

simply the main memory cycle time, t_m. However, if that access is a write, there may be contention with another processor to write to main memory. The chance of having to wait for another write to complete may be expressed as p_w, and the time penalty involved is t_w, giving an average access time for shared data as:

$$t_{shared} = p_{read}\, t_m + (1 - p_{read})(p_w t_w + t_m) \quad (1)$$

For private (non-shared) data, a cache hit is responded to in the cache cycle time, t_c, whereas a cache miss requires an additional block transfer time, t_b, to load the requested block from main memory into cache. If the hit ratio is designated h, the access time for private data is:

$$t_{private} = t_c + (1 - h)\, t_b \quad (2)$$

If the fraction of data classified as shared is f_{shared}, the average access time for this strategy may be expressed as:

$$t_{ave} = f_{shared}\,[p_{read}\, t_m + (1 - p_{read})(p_w t_w + t_m)] + (1 - f_{shared})[t_c + (1 - h)\, t_b] \quad (3)$$

3.3 Synapse

A read hit fetches data directly from the cache in t_c time. On a read miss, if another cache has a dirty copy of the block, it must be copied back to main memory before the requesting cache can read it. Assuming a dirty copy exists in another cache with probability p_{dirty}, the average time required to write back dirty blocks (including write contention) is $p_{dirty}\ (p_w t_w + t_b)$. Finally, the block may be transferred to the requesting cache in t_b time. The average access time for reads becomes:

$$t_{read} = t_c + (1 - h)[p_{dirty}\ (p_w t_w + t_b) + t_b] \quad (4)$$

A write hit can take place in t_c time if the block is already dirty in that cache. If the local state is not dirty, $1 - p_{dhere}$, it is possible the block could be shared. Since Synapse does not send invalidation signals, the updated block must be written through to main memory via the snooping bus. All other caches will see this write and invalidate their copies of the data. The probability of contention for the snooping bus may be represented p_{cont}, and the wait time due to contention t_{cont}. A write miss involves identical actions to a read miss, making the overall write access time:

$$t_{write} = t_c + (1 - h)[p_{dirty}\ (p_w t_w + t_b) + t_b] + h(1 - p_{dhere})(p_{cont}\, t_{cont} + t_b) \quad (5)$$

Equations (4) and (5) combine for an overall access time of:

$$t_{ave} = t_c + (1 - h)[p_{dirty}\ (p_w t_w + t_b) + t_b] + h(1 - p_{read})(1 - p_{dhere})(p_{cont}\, t_{cont} + t_b) \quad (6)$$

3.4 Firefly

As before, a read hit requires t_c time. On a read miss, if the data exists (clean or dirty) in another cache (p_{shared}), a direct cache-to-cache block transfer occurs in time t_{cb}. The supplying cache simultaneously updates main memory, but that step does not involve the requesting cache, and is therefore transparent to it. If no cache holds a copy, the block is forwarded from main memory directly. This leads to:

$$t_{read} = t_c + (1 - h)[p_{shared}\, t_{cb} + (1 - p_{shared})\, t_b] \quad (7)$$

A write hit takes t_c time if the block is either in state DIRTY or VALID-EXCLUSIVE. If the block is SHARED, however, the snooping bus must be obtained, and the updated word written to main memory. All caches observe the write, and update their own copies accordingly. A write miss operation responds similarly to a read miss operation: data is provided from another cache if possible, or main memory otherwise. However, data transferred from another cache is loaded in state SHARED, so the requesting cache must write the word in question to main memory, as in a write hit. This results in an average write time of:

$$t_{write} = t_c + hp_{shared}\,[p_{cont}\, t_{cont} + t_m] + (1 - h) \times [p_{shared}\ (t_{cb} + p_{cont}\, t_{cont} + t_m) + (1 - p_{shared})\, t_b] \quad (8)$$

Which may be simplified to:

$$t_{write} = t_c + p_{shared}\,[p_{cont}\, t_{cont} + t_m] + (1 - h)[p_{shared}\, t_{cb} + (1 - p_{shared})\, t_b] \quad (9)$$

Combining equations (7) and (9) gives an overall access time of:

$$t_{ave} = t_c + (1 - h)[p_{shared}\, t_{cb} + (1 - p_{shared})\, t_b] + (1 - p_{read})\, p_{shared}\ (p_{cont}\, t_{cont} + t_m) \quad (10)$$

3.5 Directory

A read hit is answered in t_c. A read miss requires checking the global directory for other copies of the data. Like the Synapse protocol, if any other cache has a dirty

copy (p_{dirty}), it must be written to main memory before the requesting cache may retrieve it. This makes the read access time:

$$t_{read} = t_c + (1-h)[t_d + p_{dirty}(p_w t_w + t_b) + t_b] \quad (11)$$

A write hit requires sending an invalidation signal to all other caches only if multiple copies exist (p_{shared}). Contention for the invalidate line is modeled as p_{cont} with wait time t_{cont}. The invalidate signal may be propagated in t_{inv} time. A write miss is handled identically to a read miss, giving:

$$t_{write} = t_c + (1-h)[t_d + p_{dirty}(p_w t_w + t_b) + t_b] + h p_{shared}(p_{cont} t_{cont} + t_{inv}) \quad (12)$$

The overall access time is therefore:

$$t_{ave} = t_c + (1-h)[t_d + p_{dirty}(p_w t_w + t_b) + t_b] + h p_{shared}(1-p_{read})(p_{cont} t_{cont} + t_{inv}) \quad (13)$$

4 Model Inputs

Equations (3), (6), (10), and (13) define the models for each of the coherence strategies. However, for these to be truly effective requires reasonable assumptions for the input parameters. The following sections justify values for these variables.

4.1 Memory Access Times

The central assumption is a main memory cycle time of 100 ns. This is roughly equivalent to the access time of typical DRAM technology [18]. Since cache memories are commonly designed with speeds 3-10 times faster than main memory, the ratio of main memory cycle time to cache cycle time was varied from 2 to 10 (cache time 50-10 ns) for one trial, with a fixed value of 5 (cache time 20 ns) for all other tests.

Current technology supports the construction of a small multiported directory equally fast as cache, as shown in [11]. Since each use of the directory will require initially checking and later updating its elements, this defines t_d equal to $2t_c$ for all test runs.

Cache and main memory block transfer times are simply the respective cycle times multiplied by the number of words per block. A typical value of four words per block was chosen, consistent with [3] and [20].

4.2 Write Contention

The probability of write contention is a function of both the percentage of writes, and the main memory write convention (CREW or CRCW). **Given that a processor is writing to memory**, the probability of contention is the likelihood of *either* of the remaining two processors also writing. Since our ideal application exhibits a uniform problem distribution, the chance of any processor writing is both equal to and independent of the others. The probability of either of two independent events occurring is expressed as [13]:

$$p(A \vee B) = p(A) + p(B) - p(A)\,p(B) \quad (14)$$

so once the chance of a single cache interfering is determined, the overall likelihood of contention from either cache may be computed by substitution into equation (14). The next several sections determine the single cache probabilities, which were transformed into overall contention chances by use of equation (14).

4.2.1 No-Cache Write Contention

The No-Cache scheme requires all writes to shared data to write-through to main memory. In CREW access, the chance of write contention by a single processor is its probability of writing shared data, $p_{shared}(1-p_{read})$, since any two simultaneous writes result in contention.

For CRCW access, contention exists only if more than one cache attempts to write to the same partition. It follows that as the number of partitions increases, the likelihood of writing to a given partition decreases. Assuming even distribution, the probability of a processor writing to a given partition may be expressed as:

$$\frac{p_{shared}(1-p_{read})}{partitions} \quad (15)$$

In either case, the wait time is either one or two main memory cycles, based on whether one or both other processors were attempting a write. One cycle time will be assumed for this model.

4.2.2 Snooping Write Contention

For all snooping strategies except Firefly, a requested block which is dirty in another cache must be written to main memory before the requesting cache may read it. This occurs with frequency $(1-h)p_{dirty}$, which is the chance for single processor write contention.

A CRCW model merely divides this probability by the number of partitions, resulting in a likelihood of

$$\frac{(1-h)\,p_{dirty}}{partitions} \quad (16)$$

For these events, the cache might wait any fraction of zero to two block transfer times, depending on when the

write collided with the block transfer. One block transfer time is assumed for this model.

4.3 Bus Contention

The probability of contention for access to the snooping bus depends on the particular strategy. Again, contention in the system may be determined from the probability of interference by a single other cache and application of equation (14).

For the Synapse protocol, access to the snooping bus is required on a write hit to a block that is not currently dirty in cache. This probability is $h(p_{read})(1-p_{dhere})$.

For the Firefly methodology, the snooping bus is used whenever shared data is being written. This chance is $(1-p_{read})(p_{shared})$.

In either case, the latency due to contention was assumed to be equal to the main memory access time.

4.4 Directory Invalidation

A cache must invalidate all other copies of a block on a write hit to data present in another cache. The probability of this occurrence for a single cache is $h(p_{shared})(1-p_{read})$. Equation (14) may then be applied.

The time to perform an invalidation (t_{inv}), is the time necessary to update the other caches in parallel, or t_c. Therefore, the wait time for contention, t_{cont}, is also t_c.

4.5 Data Sharing

Both Firefly and Directory schemes require action when data which exists in more than one cache is referenced. The probability of a block being shared when accessed, p_{shared}, is proportional to the fraction of shared data in the application, f_{shared}. The constant of proportionality represents how often potentially shared data is referenced in the application program, and whether the data is actually being shared at that time.

Both the value of f_{shared} and the proportionality constant are highly dependant on the particular application. Values for this fraction of shared data were varied from 0 to 100% for one test, with a value of 25% used in all other simulations. The constant was assigned a value of 0.79, consistent with an average value observed in large-cache traces used in [16].

The probability of a block being dirty in another cache when it is referenced, p_{dirty}, is also proportional to f_{shared}. This constant (0.16) was taken from the same source.

4.6 System Inputs

Reasonable input values for the free variables established in these models were either taken directly from, or adapted from [20], [3], and [16]. Table 5 lists the inputs to the mathematical model, and the values used in calculations.

Input Parameter	Value	Range
h	.95	.75 - 1.0
p_{read}	.75	.50 - 1.0
f_{shared}	.25	0 - 1.0
partitions	16	1 - 128
ratio MM/CM time	5	2 - 10
words per block	4	n/a
p_{dhere}	.3	n/a

Table 5 System Parameters

5 Results

Five different experiments were performed for each of the protocols described above. Each simulation varied one parameter across a range while keeping all other input variables constant. Since the choice of CREW vs CRCW has a negligible effect on all protocols, graphs of the results will compare only the CRCW versions of each methodology.

5.1 Varying Percent Read Accesses

This demonstration varied the percentage of read accesses from 50% to 100% to illustrate the effects on coherence protocol overhead as more writing is required.

Figure 2 compares the CREW and CRCW versions of the No-Cache, Synapse, and Directory protocols. The most prominent feature is the virtually identical plots for both the CREW and CRCW implementations. The Synapse and Directory strategies show a variance of only 0.002 ns; No-Cache protocols are within 1 ns for all read fractions above 0.63. Since these protocols write to main memory so infrequently, the chance of two processors simultaneously attempting to do so is negligible, thus making the chance of CREW (and CRCW) contention negligible.

However, for the No-Cache strategy at low read percentages, the number of writes to main memory is **not** negligible, and results in CREW contention. The CRCW version uses sufficient partitioning of main memory to ensure that the likelihood of simultaneous writes accessing

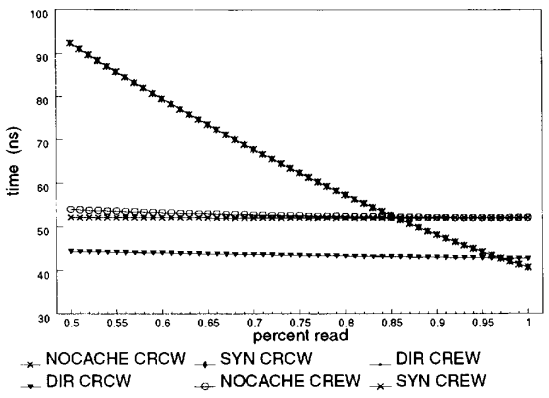

Figure 2 CREW vs CRCW Methodologies

the same partition is minuscule.

Since the deviation between CREW and CRCW is insignificant, only CRCW results will be reported for the remaining demonstrations.

Figure 3 illustrates the performance of all CRCW techniques and the Firefly protocol with respect to read percentage. The No-Cache methodology exhibits the least change over the range in question since read and writes are handled identically by the protocol. The only difference occurs when writing shared data, as contention may occur for main memory. Since we have shown contention to be insignificant, it follows that there is no impact on the overall access time. Directory's access time is also reasonably constant, since the only impact occurs on write hits to shared data, as an invalidation signal must be sent. Write hits to shared data are rare, and invalidation times are short, so little effect is observed.

Firefly's performance improves since all writes to shared data must be passed on to main memory, but Synapse shows significantly more improvement, since write hits to clean data require the entire **block** to be copied to main memory. As writes decrease, this substantial time penalty is mitigated. Note that this protocol is by far the most sensitive to changes in read percentage.

5.2 Varying the Fraction of Shared Data

Most coherence strategies rely on invalidating copies of shared data when they are updated. It follows that performance can depend on the fraction of shared data references. This fraction depends on the applications in question, so a wide range of values is possible. This simulation varies the fraction of shared data from 0 to 1.0 with results graphed in Figure 4.

Since No-Cache requires all shared data references to access main memory rather than cache, there is a dramatic, linear relationship between the fraction of shared data and access time. This technique would only be useful for applications with little shared data.

Synapse is affected by changes in the fraction of shared data only through indirect means: as more data is shared, the chance of a requested block existing dirty in another cache increases. This elevates the possibility of having to wait for the dirty block to be copied to main memory before being loaded into the requesting cache. Although this protocol is very stable for changes in sharing, its overhead far outweighs its stability.

The Directory coherence strategy is affected by the same indirect means as Synapse. Additionally, write hits to shared data necessitate an invalidation signal. Invalidation latency causes a 19.6% increase in access time over the entire range — a relatively small deviation compared to Firefly.

Since Firefly allows direct cache-to-cache transfers of shared data but requires main memory transfers for private, it would be expected that times would improve as more data is shared. Instead, Firefly times **increase** by 35.4%. Improvement from cache-to-cache transfers is

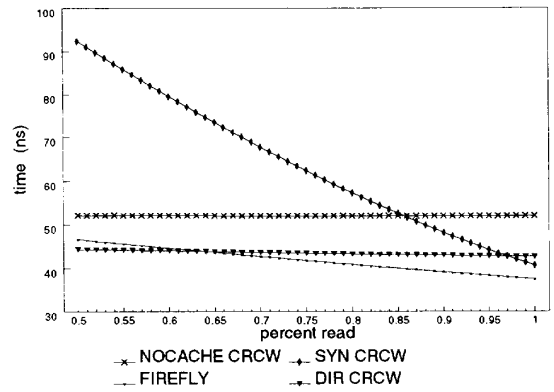

Figure 3 Performance vs Read Percentage

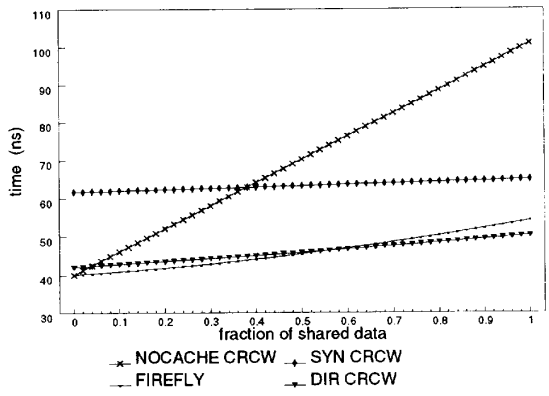

Figure 4 Performance vs Fraction of Shared Data

minimal, since they only occur on (infrequent) cache misses. The elevation of access times occurs since writes to shared data compel sending each word to memory via the snooping bus, resulting in contention and main memory latency. This effect far outweighs the slight improvement of cache-to-cache transfers.

For low data sharing applications, Firefly is the faster protocol, however fractions of shared data above 0.56 are more suited to the Directory scheme.

5.3 Varying the Number of Partitions

When CRCW write access is employed for main memory, simultaneous writes are permitted only when targets are in different partitions. Thus, the number of partitions will affect the likelihood of write collisions (for CRCW).

Since it was shown in Section 5.1 that the choice of CREW or CRCW is insignificant, it is expected the number of partitions used for CRCW is also immaterial. In general, this is indeed the case, however No-Cache writes to main memory enough to create a few collisions. Nevertheless, increasing the number of partitions results in only a 0.92% improvement in access time. Note that an 0.87% improvement is indicated after the first 16 partitions, and a 0.9% refinement after 32 partitions. Using more than 32 partitions is incrementally useful, at best, as shown in Figure 5.

It is useful to note that the assumption of an infinite cache eliminates writing blocks back to main memory upon page replacement. In a real system, these writes would increase the number of collisions. Thus, the benefit of partitions would be more evident if a replacement policy is assumed.

5.4 Varying the Cache Cycle Time

Speed of cache memory is an important consideration in cache performance. The ratio of main memory to cache memory can vary tremendously between computer systems. This model varies the ratio from 2 to 10 to examine the effects of cache speed on average access time for each protocol, with results illustrated in Figure 6.

As a reference to the other protocols, Synapse accesses the cache exactly once for each memory reference. It shows a 43.3% improvement throughout the test.

No-Cache shows less improvement since it accesses the cache only when private data is referenced. Since the majority of data was assumed private, only a small impact is observed, giving a refinement of 42.1%.

Firefly experiences a significant improvement (56.8%) based on its use of cache-to-cache transfers, although Directory's 58.7% refinement is slightly better. Directory's advantage is that directory access time and

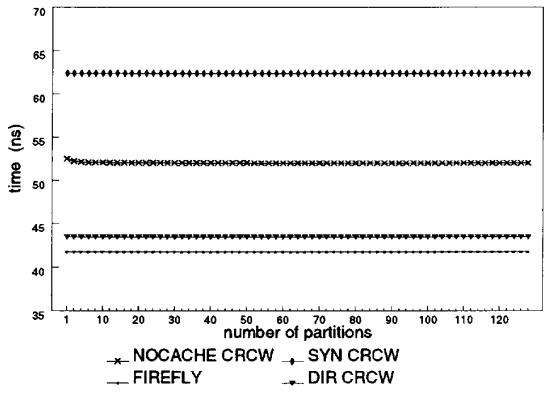

Figure 5 Performance vs Number of Partitions

cache invalidation time are assumed to be equal to cache time; as cache time decreased, so did the directory and invalidation latencies. In spite of this advantage, the Directory scheme doesn't outperform Firefly until the ratio is at least 29. Such a ratio corresponds to a cache time of 3.4 ns — far beyond today's technology.

5.5 Varying the Hit Ratio

The original idea behind cache memory systems was to allow the processor to retrieve data at or near the speed it was capable of processing. The concepts of *bandwidth matching* and *memory hierarchy* enabled processors to access data in a fast memory rather than having to access slow main memory. However, the fast memory would be useless unless the needed information was in the cache. Thus the hit ratio became an important metric of cache performance. This analysis varied the hit ratio from 0.75 to 1.0, as shown in Figure 7.

Since cache performance is strongly dependent on hit ratio, it is not surprising that all coherence protocols showed at least 68% improvement. The Directory

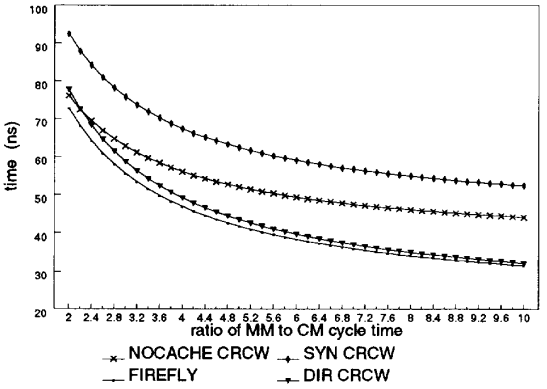

Figure 6 Performance vs Cache Cycle Time

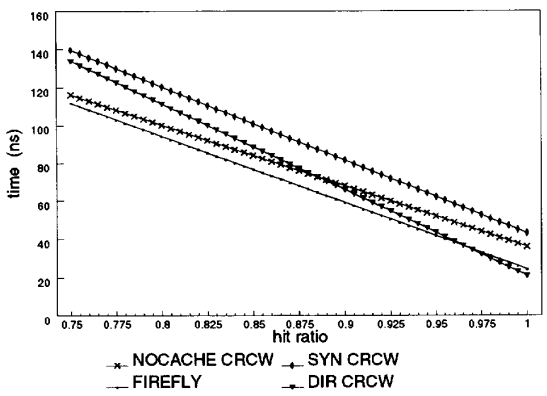

Figure 7 Performance vs Hit Rate

scheme showed the greatest improvement with 84.4%, compared to 78.2% for Firefly, 69.1% for Synapse, and 68.9% for No-Cache.

It is well known that implementing cache coherence causes hit ratios to decrease due to block invalidations [7], and that parallel programs in general have lower hit ratios [8]. It follows that intelligent selection of a cache coherence strategy must consider the system's expected hit ratio.

For hit ratios above 0.96, the Directory scheme shows the fastest performance, with Firefly fastest below. However, on an 8% decrease in hit rate (0.88), the No-Cache scheme also surpasses Directory. This implies that a highly optimized cache performs significantly better with a complex coherence strategy (Firefly or Directory), whereas intricate protocols are superfluous with less optimized caches.

6 Conclusions

Overall, the Firefly coherence scheme seemed to consistently exhibit the fastest times. However, to support its cache-to-cache transfers requires extensive hardware: dedicated paths must exist between each pair of caches. Such hardware costs will become enormous as the number of processors is increased, making Firefly less scalable than other techniques.

The Directory protocol was nearly as fast as Firefly, but much more scalable. As main memories are designed with more ports, so too will small, fast directory memories. Although the cost of having a fast multiported memory for the directory is significant, only a single memory with the same number of ports as main memory is required, making this strategy much more scalable than Firefly.

No-Cache has the advantage of extremely low overhead — no additional hardware is required to implement this protocol, nor are there significant time penalties to perform nonproductive coherence tracking. This solution might be an appropriate option for systems with little coherence problems (*i.e.* high read percentages, little shared data).

Synapse overhead is staggering since it requires writing an entire **block** to main memory on a write hit to clean data. If this protocol implemented a write of only the **word** in question, its performance would be near the other strategies.

7 Future Work

Trace-driven simulations are being conducted to corroborate the findings of analytical analysis. In addition, elements not easily modeled will be included to produce real-system values. Probabilities will be represented as Poisson distributions, and the effects of block replacement strategy and block size will be simulated using finite caches. We also intend to investigate the effect of increasing the number of memory ports on coherence strategies, as well as the hardware trade-offs involved.

References

[1] Agarwal, Simoni, Hennessy, Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*. 16(2):280-289, May 1988.

[2] Archibald, Baer. An economical solution to the cache coherence problem. In *Proceedings of the International Symposium on Computer Architecture*, pages 355-362, 1984.

[3] Archibald, Baer. Cache coherence protocols: evaluation using a microprocessor simulation model, *ACM Transactions on Computer Systems*, 4(4):273-298, November 1986.

[4] Bertoni, Wang. Multiple interleaved bus architectures. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 1-8, August 1991.

[5] Censier, Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27(12):1112-1118, December 1978.

[6] Cheong, Veidenbaum. A Cache Coherence Scheme With Fast Selective Invalidation. In *Proceedings of the 15th International Symposium on Computer Architecture*. 16(2):299-307, May 1988.

[7] Dubois, Briggs. Effects of Cache Coherency in Multiprocessors. In *Proceedings of the 9th International Symposium on Computer Architecture*. 10(3):299-308, April 1982.

[8] Eggers, Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *ASPLOS-III Proceedings, Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257-270, April 1989.

[9] Gupta, Weber. Analysis of cache invalidation patterns in multiprocessors. In *ASPLOS-III Proceedings, Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243-256, April 1989.

[10] Hwang, Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, Inc., 1984, pages 519-525.

[11] Integrated Device Technology. *High-Speed 2K x 8 Four-Port Static RAM, Advance Information*. Integrated Device Technology, January 1989.

[12] Katz, Eggers, Wood, Perkins, Sheldon. Implementing a Cache Consistency Protocol. In *Proceedings of the 12th International Symposium on Computer Architecture*. 13(3):276-283, June 1985.

[13] Milton, Arnold. *Probability and Statistics in the Engineering and Computing Sciences*. McGraw-Hill, Inc., 1986, pages 21-34.

[14] Min, Baer, Kim. An Efficient Caching Support for Critical Sections in Large-Scale Shared Memory Multiprocessors. In *Proceedings of the International Conference on Supercomputing*. 18(3):34-47, June 1990.

[15] O'Krafka, Newton. An empirical evaluation of two memory-efficient directory methods. In *Proceedings of the International Symposium on Computer Architecture*. 18(2):138-147, June 1990.

[16] Owicki, Agarwal. Evaluating the performance of software cache coherence. In *ASPLOS-III Proceedings, Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 230-242, April 1989.

[17] Stodleck. The IDT FourPort™ RAM facilitates multiprocessor designs. *Application Note AN-43*, pages 1-13, 1989.

[18] Texas Instruments. *MOS Memory Commercial and Military Specifications Data Book*. Texas Instruments, Inc. 1993.

[19] Thapar, Delagi. Cache coherence for large scale shared memory multiprocessors. *Computer Architecture News*, 19(1):114-119, March 1991.

[20] Vernon, Lazowska, Zahorjan. An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache Consistency Protocols. In *Proceedings of the 15th Annual Symposium on Computer Architecture*. 16(2):308-315, May 1988.