Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model

JAMES ARCHIBALD and JEAN-LOUP BAER
University of Washington

Using simulation, we examine the efficiency of several distributed, hardware-based solutions to the cache coherence problem in shared-bus multiprocessors. For each of the approaches, the associated protocol is outlined. The simulation model is described, and results from that model are presented. The magnitude of the potential performance difference between the various approaches indicates that the choice of coherence solution is very important in the design of an efficient shared-bus multiprocessor, since it may limit the number of processors in the system.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*cache memories*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*multiple-instruction-stream, multiple-data-stream processors (MIMD)*; C.4 [**Computer Systems Organization**]: Performance of Systems—*measurement techniques*; *modeling techniques*; D.4.2 [**Operating Systems**]: Storage management—*distributed memories*

General Terms: Design, Performance

Additional Key Words and Phrases: Cache coherence, shared-bus multiprocessor, simulation

1. INTRODUCTION

There is currently considerable interest in the computer architecture community on the subject of shared-memory multiprocessors. Proposed multiprocessor designs often include a private cache for each processor in the system, which gives rise to the *cache coherence problem.* If multiple caches are allowed to have copies simultaneously of a given memory location, a mechanism must exist to ensure that all copies remain consistent when the contents of that memory location are modified. In some systems, a software approach is taken to prevent the existence of multiple copies by marking shared blocks as not to be cached, and by restricting or prohibiting task migration. An alternate approach is to allow all blocks to be cached by all processors and to rely on a *cache coherence protocol* (between the cache controllers and, in some cases, memory controllers) to maintain consistency.

Several such protocols have been proposed or described—some suitable for a general interconnection network [1, 2, 14, 16] and some specifically for a

This work was supported in part by NSF grants MCS-8304534 and DCR-8503250.
Authors' address: Department of Computer Science, University of Washington, Seattle, WA 98195.

shared-bus [5, 6, 8–11, 15]. Shared-bus protocols differ substantially from protocols for general networks because first, they depend on each cache controller observing the bus transactions of all other processors in the system, taking appropriate actions to maintain consistency, and second, the state of each block in the system is encoded in a *distributed* way among all cache controllers. Cache controllers that observe the bus traffic for coherence purposes are called *snooping* cache controllers.

In this paper we examine several distributed hardware-based protocols for shared-bus multiprocessors and evaluate their relative performance on the basis of a simulation model. All of the schemes discussed in this paper require snooping cache controllers. Although a number of different hardware implementations for such cache controllers exist, each with a different level of performance, it is our goal in this paper to identify the relative performance of the protocols independent of differences in implementation. For this reason, we evaluate the schemes assuming identical processors and caches, except for the necessary differences in the cache controller to support the protocol. We begin with a brief description of the schemes to be analyzed and then describe the simulation model used. Simulation results are then presented and discussed.

2. CACHE COHERENCE PROTOCOLS

In a shared-bus multiprocessor, the bus becomes the limiting system resource with even a moderate number of processors. The key to maximizing overall system performance is minimizing the bus requirements of each individual processor. The addition of a private cache for each processor can greatly reduce the bus traffic, since most references can then be serviced without a bus transaction. Bus requirements of the caches can be further reduced by choosing a write-back (also called copy-back) main memory update policy instead of a write-through approach [12]. (In write-through, stores are immediately transmitted to main memory; write-back initially modifies only the cache with the change reflected to main memory when the block is removed from the cache.) All of the schemes considered here use a form of write-back. However, we shall simulate a write-through mechanism for comparison purposes.

With few exceptions (e.g., Firefly [15] and Dragon [9]), all proposed solutions enforce consistency by allowing any number of caches to read a given block but allowing only one cache at a time permission to write the block. Unlike efficient solutions for general interconnection networks requiring information in a global table, shared-bus solutions maintain coherence on the basis of information maintained locally at each cache. Each cache controller listens to transactions on the bus and takes actions, if necessary (depending on the type of transaction and the local state of the block), to maintain the consistency of those blocks of which it has copies. For each bus transaction, the snooping cache controller must determine whether it has a copy of the block by attempting to match the block address observed on the bus with the address in the cache directory. If there is a single copy of the cache directory, each attempted match will require a cache cycle, during which time the cache is unable to service processor memory requests. A far more efficient alternative is to provide the controller with a duplicate copy

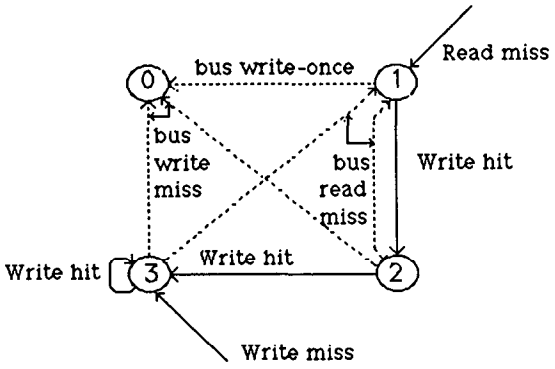

Fig. 1. Write-once transition diagram.

of the cache directory, allowing all unsuccessful attempts to match to be completed without affecting processor performance.

Each cache coherence protocol consists of a specification of possible block states in the local caches and the actions that are to be taken by the cache controller as certain bus transactions are observed. To outline the protocols that we examine in this paper, consider the essential actions of each scheme in the following four cases: read hit, read miss, write hit, and write miss. The case of read hit is easily dealt with—in all schemes the requested data are returned immediately to the processor with no action necessary by the protocol. Differences in the other three cases are outlined below. States are written in capital letters. (See also the accompanying state transition diagrams.)

2.1 Write-Once

Chronologically the first scheme described in the literature [6], Goodman's write-once scheme was designed for single-board computers using Multibus. The requirement that the scheme work with an existing bus protocol was a severe restriction but one that results in implementation simplicity. In the write-once scheme, blocks in the local cache can be in one of four states: INVALID, VALID (not modified, possibly shared), RESERVED (not needing a write-back, but guaranteed the only copy in any cache), and DIRTY (written more than once and the only copy in any cache) (see Figure 1). Blocks selected for replacement in the cache need to be written back to main memory only if in the DIRTY state.

The scheme works as follows:

(1) Read miss. If another copy of the block exists that is in state DIRTY, the cache with that copy inhibits the memory from supplying the data and supplies the block itself, as well as writing the block back to main memory. If no cache has a DIRTY copy, the block comes from memory. All caches with a copy of the block set their state to VALID.
(2) Write hit. If the block is already DIRTY, the write can proceed locally without delay. If the block is in state RESERVED, the write can also proceed without delay, and the state is changed to DIRTY. If the block is in state VALID, the word being written is written through to main memory (i.e., the bus is obtained, and a one-word write to the backing store takes place) and the local state is set to RESERVED. Other caches with a copy of that block (if any) observe the bus write and change the state of their block copies to INVALID. If the block is replaced in state RESERVED, it need not be written back, since the copy in main memory is current.
(3) Write miss. Like a read miss, the block is loaded from memory, or, if the block is DIRTY, from the cache that has the DIRTY copy, which then invalidates its copy. Upon seeing the write miss on the bus, all other caches with the block invalidate their copies. Once the block is loaded, the write takes place and the state is set to DIRTY.

2.2 Synapse

This approach was used in the Synapse N + 1, a multiprocessor for fault-tolerant transaction processing [5]. The N + 1 differs from other shared bus designs considered here in that it has two system buses. The added bandwidth of the extra bus allows the system to be expanded to a maximum of 28 processors. Another noteworthy difference is the inclusion of a single-bit tag with each cache block in main memory, indicating whether main memory is to respond to a miss on that block. If a cache has a modified copy of the block, the bit tells the memory that it need not respond. This prevents a possible race condition if a cache does not respond quickly enough to inhibit main memory from responding. Cache blocks are in one of the following states: INVALID, VALID (unmodified, possibly shared), and DIRTY (modified, no other copies) (see Figure 2). Only blocks in state DIRTY are written back when replaced. Any cache with a copy of a block in state DIRTY is called the *owner* of that block. If no DIRTY copy exists, memory is the owner. The Synapse coherence solution is the following:

(1) Read miss. If another cache has a DIRTY copy, the cache submitting the read miss receives a negative acknowledgement. The owner then writes the block back to main memory, simultaneously resetting the bit tag and changing the local state to INVALID. The requesting cache must then send an additional miss request to get the block from main memory. In all other cases the block comes directly from main memory. Note that the block is always supplied by its owner, whether memory or a cache. The loaded block is always in state VALID.

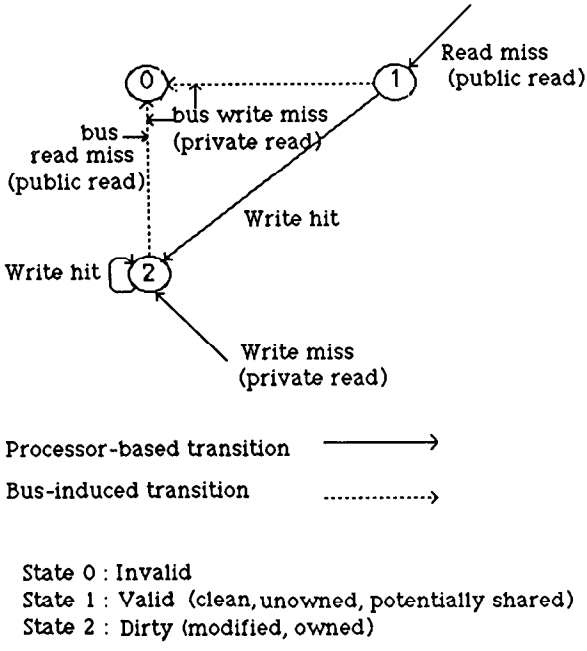

Fig. 2. Synapse transition diagram.

(2) Write hit. If the block is DIRTY, the write can proceed without delay. If the block is VALID, the procedure is identical to a write miss (including a full data transfer) since there is no invalidation signal.

(3) Write miss. Like a read miss, the block always comes from memory—if the block was DIRTY in another cache, it must first be written to memory by the owner. Any caches with a VALID block copy set their state to INVALID, and the block is loaded in state DIRTY. The block's tag in main memory is set so that the memory ignores subsequent requests for the block.

2.3 Berkeley

This approach is to be implemented in a RISC multiprocessor currently being designed at the University of California at Berkeley [8]. The scheme is similar to the Synapse approach, with two major differences: It uses direct cache-to-cache transfers in the case of shared blocks, and dirty blocks are not written back to memory when they become shared—requiring one additional state. The following states are used: INVALID, VALID (possibly shared and not modified), SHARED–DIRTY (possibly shared and modified), and DIRTY (no other copies in caches and modified) (see Figure 3). A block in either state SHARED–DIRTY or DIRTY must be written back to main memory if it is selected for replacement. A block in state DIRTY can be in only one cache. A block can be in state SHARED–DIRTY in only one cache, but it might also be present in state VALID in other caches. Like the Synapse protocol, Berkeley uses the idea of ownership—the cache that has the block in state DIRTY or SHARED–DIRTY is the owner

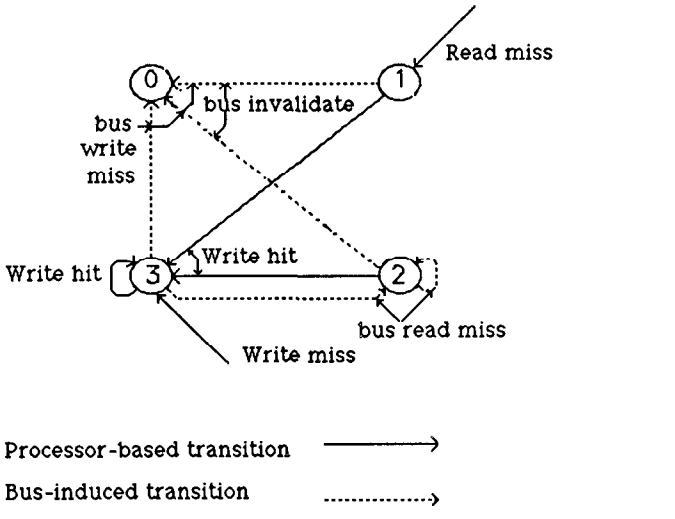

Fig. 3. Berkeley transition diagram.

of that block. If a block is not owned by any cache, memory is the owner. The consistency solution is the following:

(1) Read miss. If the block is DIRTY or SHARED–DIRTY, the cache with that copy must supply the block contents directly to the other cache and set its local state to SHARED–DIRTY. If the block is in any other state or not cached, it is loaded from main memory. In any case, the block state in the requesting cache is set to VALID. Note that the block always comes directly from its owner.

(2) Write hit. If the block is already DIRTY, the write proceeds with no delay. If the block is VALID or SHARED–DIRTY, an invalidation signal must be sent on the bus before the write is allowed to proceed. All other caches invalidate their copies upon matching the block address, and the local state is changed to DIRTY in the originating cache.

(3) Write miss. Like a read miss, the block comes directly from the owner. All other caches with copies change the state to INVALID and the block in the requesting cache is loaded in state DIRTY.

2.4 Illinois

This approach [10] assumes that missed blocks always come from other caches, if any copies are cached, and from memory if no cache has a copy, and it is also assumed that the requesting cache will be able to determine the source of the block. Each time that a block is loaded it can therefore be determined whether or not it is shared. This information can significantly improve the system

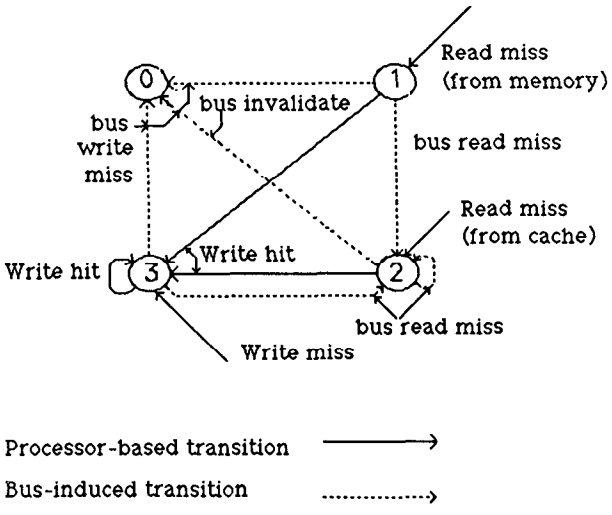

Fig. 4. Illinois transition diagram.

performance since invalidations for write hits on unmodified private blocks can be entirely avoided. The scheme has the following four states for cached blocks: INVALID, VALID-EXCLUSIVE (not modified, the only copy in caches), SHARED (not modified, possibly other copies cached), and DIRTY (modified and the only cached copy) (see Figure 4). Blocks are written back at replacement only if they are in state DIRTY. The scheme works as follows:

(1) Read miss. If any other cache has a copy of the block, it puts it on the bus. If the block is DIRTY, it is also written to main memory at the same time. If the block is SHARED, the cache with the highest priority will succeed in putting the block on the bus. All caches with a copy of the block will observe the miss and set their local states to SHARED, and the requesting cache sets the state of the loaded block to SHARED. If the block comes from memory, no other caches have the block, and the block is loaded in state VALID-EXCLUSIVE.

(2) Write hit. If the block is DIRTY, it can be written with no delay. If the block is VALID-EXCLUSIVE, it can be written immediately with a state change to DIRTY. If the block is SHARED, the write is delayed until an invalidation signal can be sent on the bus, which causes all other caches with a copy to set their state to INVALID. The writing cache can then write to the block and set the local state to DIRTY.

(3) Write miss. Like a read miss, the block comes from a cache, if any cache has a copy of the block. All other caches invalidate their copies, and the block is loaded in state DIRTY.

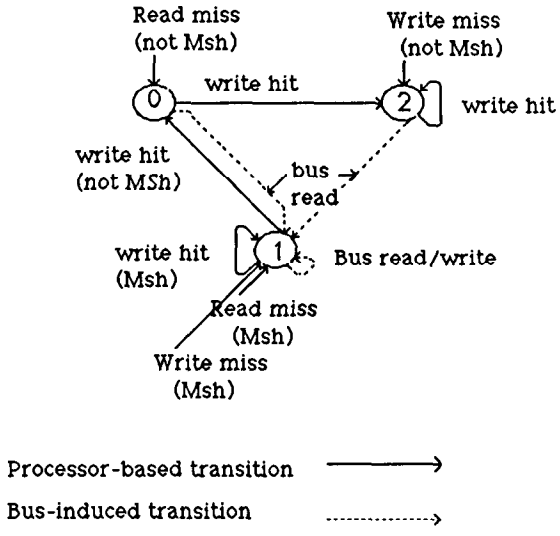

Fig. 5. Firefly transition diagram.

2.5 Firefly

This scheme is used in the Firefly [15], a multiprocessor workstation currently being developed by Digital Equipment Corporation. Possible states for blocks in local caches are: VALID–EXCLUSIVE (not modified, only copy in caches), SHARED (not modified, possibly other caches with a copy), and DIRTY (modified, only copy in caches) (see Figure 5). Blocks in state DIRTY are the only ones that are written back to memory at replacement. The main difference between this scheme and those previously discussed is that multiple writers are permitted—the data for each write to a shared block are transmitted to each cache and to the backing store. As a result, this scheme never causes an invalidation, and so the INVALID state is not included in this description. There is a special bus line used to detect sharing, which we refer to as the SharedLine. The protocol is described as follows:

(1) Read miss. If another cache has the block, it supplies it directly to the requesting cache and raises the SharedLine. All caches with a copy respond by putting the data on the bus—the bus timing is fixed so that they all respond in the same cycle. All caches, including the requesting cache, set the state to SHARED. If the owning cache had the block in state DIRTY, the block is written to main memory at the same time. If no other cache has a copy of the block, it is supplied by main memory, and it is loaded in state VALID–EXCLUSIVE.

(2) Write hit. If the block is DIRTY, the write can take place without delay. If the block is in state VALID-EXCLUSIVE, the write can be performed immediately and the state is changed to DIRTY. If the block is in state SHARED, the write is delayed until the bus is acquired and a write-word to main memory can be initiated. Other caches with the block observe the write-word on the bus and take the new data and overwrite that word in their copy of the block. In addition, these other caches raise the SharedLine. The writing cache can determine whether sharing has stopped by testing this line. If it is not raised, no other cache has a copy and writes need no longer be broadcast—allowing a state change to VALID-EXCLUSIVE (and then to DIRTY on the next local write). If the line is high, sharing continues and the block remains in state SHARED.

(3) Write miss. As with a read miss, the block is supplied by other caches, if any other caches have a copy. The requesting cache determines from the SharedLine whether or not the block came from other caches. If it came from memory, it is loaded in state DIRTY and written to without additional overhead. If it came from a cache, it is loaded in state SHARED and the requesting cache must write the word to memory. Other caches with a copy of the block will take the new data and overwrite the old block contents with the new word.

2.6 Dragon

The Dragon [9] is a multiprocessor being designed at Xerox Palo Alto Research Center. The coherence solution employed is very similar to the Firefly scheme described above. The scheme employs the following states for blocks present in the cache: VALID-EXCLUSIVE (only copy in caches, but not modified), SHARED-DIRTY (write-back required at replacement), SHARED-CLEAN, and DIRTY (only copy in caches and modified) (see Figure 6). As with the Firefly, the Dragon scheme allows multiple writers, but, unlike the Firefly, writes to shared blocks are not immediately sent to main memory, only to other caches that have a copy of the block. This necessitates the addition of the SHARED-DIRTY state, implying that the block may be shared, and that it is modified with respect to the backing store, and that the cache with this copy is responsible for updating memory when the block is replaced. When a block is actually shared, the last cache to write it, if any, will have the block in state SHARED-DIRTY. All other caches with a copy will have the block in state SHARED-CLEAN. As with the Firefly scheme, the INVALID state is not included, and a SharedLine on the bus is assumed. The protocol works as follows:

(1) Read miss. If another cache has a DIRTY or SHARED-DIRTY copy, that cache supplies the data, raises the SharedLine, and sets its block state to SHARED-DIRTY. Otherwise, the block comes from main memory. Any caches with a VALID-EXCLUSIVE or SHARED-CLEAN copy raise the SharedLine and set their local state to SHARED-CLEAN. The requesting cache loads the block in state SHARED-CLEAN if the SharedLine is high; otherwise, it is loaded in state VALID-EXCLUSIVE.

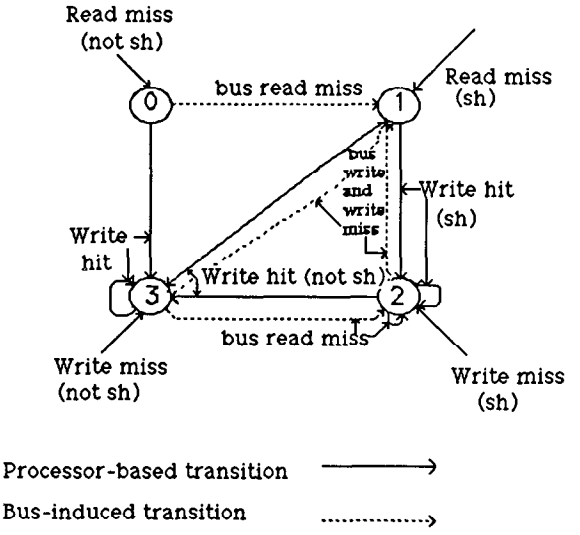

Fig. 6. Dragon transition diagram.

(2) Write hit. If the block is DIRTY, the write can take place locally without delay. If the block is in state VALID–EXCLUSIVE, the write can also take place immediately with a local state change to DIRTY. Otherwise, the block is SHARED–CLEAN or SHARED–DIRTY and a bus-write must take place. When the bus is obtained, the new contents of the written word are put on the bus and read by all caches with a copy of that block, which take the new data and overwrite that word of their copy of the block. Additionally, each such cache sets the local state of the block to SHARED–CLEAN and raises the SharedLine, indicating that the data are still shared. By observing this line on the bus, the cache performing the write can determine whether other caches still have a copy and hence whether further writes to that block must be broadcast. If the SharedLine is not raised, the block state is changed to DIRTY; else it is set to SHARED–DIRTY. Note that the single-word write does not go to main memory.

(3) Write miss. As with a read miss, the block comes from a cache if it is DIRTY or SHARED–DIRTY and from memory otherwise. Other caches with copies set their local state to SHARED–CLEAN. Upon loading the block, the requesting cache sets the local state to DIRTY if the SharedLine is not raised. If the SharedLine is high, the requesting cache sets the state to SHARED–DIRTY and performs a single-word bus write to broadcast the new contents.

3. SIMULATION MODEL

The order in which the protocols were described in the previous section was purposely chosen to show, qualitatively, either a more complex protocol or an increased reliance on the bus intelligence. The primary goal of this study is to give some quantitative measure of the efficiency of the protocols with the main metrics, defined more precisely later, being related to the number of processors that can share a common bus without reaching saturation of the system. For this purpose we use a simulation model, described below, rather than analytical models, which could not capture the subtle differences between some of the protocols. The simulation model is driven by synthetic reference streams rather than by actual traces, since no such multiprocessor traces exist. Such traces could be created, but they would be as artificial as the method that we have employed.

The first step in the simulation, written in Simula, was the creation of a basic multiprocessor model. To this basic model, protocol-specific additions were made, creating a different version for each scheme evaluated. As our intent was to evaluate the protocols themselves and not implementations thereof, we assume all protocol-independent system parameters to be identical. Thus the workload is the same: For each simulation run the reference stream of a processor is identical for all schemes and depends only on the seed variable. We also assume identical system configurations: All schemes are evaluated with one bus (although the Synapse N + 1 actually has two) and with caches with two copies of the cache directory (although not actually implemented in the Firefly and Dragon workstations). This added directory allows the bus-watching logic to attempt block address matches without affecting the performance of the cache, except in the case of a successful match when action needs to be taken.

3.1 Multiprocessor Model

The basic model consists of a Simula process for each processor, a process for each cache, and a single process for the system bus. Each processor, after performing useful work for some w cycles (picked from some distribution), generates a memory request, puts that request into the service queue of its cache, and waits for a response, during which time no work is done. Processor utilization is measured by the ratio of time spent doing useful work to the total run time. System performance is measured by the total sum of processor utilization in the system.

Each cache services memory requests from its processor by determining whether the requested block is present or absent, or, more precisely, whether the request can be serviced without a bus transaction. If so, after one cycle the cache sends the processor a command to continue. If a bus transaction is required, a bus request is generated and inserted into the service queue of the bus. The cache sends the processor a command to continue only upon completion of the bus transaction.

The cache can also receive commands from the bus process relating to actions that must be performed on blocks of which it has copies. Such commands have higher priority for service by the cache than processor memory requests. In a multiprocessor, this is equivalent to matching a block address on a bus transaction

and halting the service of processor requests to take action as specified by the protocol. After that action is completed, the cache is free to respond to processor requests. Note that such a match can occur only in the case of actual data sharing and hence is infrequent.

The bus process receives service requests from all caches and services them in first-in, first-out order. Requests are one of four types: read miss, write miss, write-back of a dirty block, and (depending on the scheme) a request for write permission, an invalidation signal, or a broadcast of the new value of a word—all dealing with write hits on unmodified shared blocks. Conceptually, the bus process includes the added cache logic responsible for matching addresses and so can determine the location of all cached copies of shared blocks. If, in servicing one of the four types of requests listed above, the bus process determines that other caches need to supply the data (if the block is requested elsewhere and is dirty), or that they need to change the local state (e.g., invalidate their copy on a shared-block write), commands are sent to the appropriate caches. When the transaction is complete, the bus signals the cache to continue.

3.2 Workload Model

The choice of workload model was viewed as critical, since it determines the nature of data sharing, and since the performance of all coherence solutions is known to depend heavily on the level of sharing. The model selected is similar to one developed in [3], although it has been extended to reflect locality of shared references. The simulation parameters and ranges used are summarized in Table I.

The reference stream of each processor is viewed as the merging of two reference streams—one being references to shared blocks and the other references to private blocks. Each time a memory reference is called for, the processor generates a reference to a shared block with probability *shd* and a reference to a private block is generated with probability $1 - shd$. Similarly, the probability that the reference is a read is *rd* and the probability that it is a write is $1 - rd$.

If the request is to a private block, it is a hit with probability h and a miss with probability $1 - h$. If the request is a write hit, the block is already modified with probability *wmd* and the block is not yet modified (in the local cache) with probability $1 - wmd$. Private blocks are never present in other caches by definition. Note that the workload model for private blocks reflects steady-state behavior and not behavior including a *cold start* (i.e., the cache is already loaded with most of the blocks that it will access in the next several references and the hit ratio has leveled out).

In the simulation model an explicit representation is chosen for shared blocks, whereas the representation of private block references is probabilistic. For private blocks the reference nature is unchanged from the uniprocessor case, and it is therefore possible to use existing uniprocessor cache measurements to reflect actions resulting from private block references. Shared block references, however, are not sufficiently well documented to use a probabilistic approach. To reflect the differences between the protocols, a probabilistic shared block model would necessarily include such information as the probability that a block is present in another cache and modified on a write miss in the local cache. In the absence of

Table I. Summary of Parameters and Ranges

Parameter	Range
shd	0.1–5%
rd	70–85%
h	95–98%
1-*wmd*	1.75–5.26%
md	30–40%
w	Uniform [0..5]
Main memory cycle time	Four cache cycles
Block size	Four words
Cache size	2–16 kbytes
Number of shared blocks	16–1024
Number of processors	1–15

such information, all references to shared blocks in our model include a specific block number, and actions are taken by the cache controllers on the basis of the actual state of that block at that point in time.

If the request is to a shared block, the block number of the reference is determined using a least recently used (LRU) stack (unique for each processor). The probability of referencing the blocks near the top is significantly higher than those near the bottom. After each reference the most recently referenced block is placed at the top of the stack, with the others shifting down one position. The LRU stack, used to reflect locality of shared block references, is initialized uniquely for each processor in such a way that the average depth over all stacks is approximately the same for each block. To service a shared block request, the cache determines from a local table (needed for the simulation but, of course, not included in the actual implementation) whether the requested block is present, and whether a bus request must be generated (determined by the coherence protocol). Note that references to shared blocks (in the simulation) are not necessarily references to blocks that are actually present in other caches. Hence the percentage of references to shared blocks and the amount of actual sharing can be quite different.

If a cache miss occurs, either for a shared block or for a private block, a block must be ejected to make room for the new block. The probability that a shared block is selected is equal to the percentage of blocks in the cache that are shared blocks at that point in time. If the selected block is private, it is modified and needs to be written back with probability *md*; with probability 1 − *md* it has not been modified (and hence no action need be taken). If a shared block is chosen for replacement, one of those present in the cache is chosen at random. The local state of that particular block determines whether or not it is to be written back. The presence tables and local state are changed to indicate that it is no longer present in this cache—following a write-back, if any. If a write-back is required of either shared or private blocks, it is completed before the missed block is loaded.

The probabilities *md*, *wmd*, and *rd* are not independent. The probability that a block is dirty when it is replaced (*md*) is equal to the probability that it was loaded on a write miss plus the probability that it was loaded on a read miss and

later modified. The probability of loading on a read miss (or write miss) can be approximated by the percentage of read requests (or write requests), assuming that the miss ratios on reads and writes are nearly identical to the overall miss ratio. Using this approximation, if x is the percentage of blocks loaded on a read miss that are eventually modified, then

$$md = (1 - rd) + x(rd).$$

In steady state, the probability of writing to an unmodified block present in the cache must equal the probability of loading a clean block into the cache times the percentage of blocks loaded on a read miss and eventually modified (or x above). That is,

$$(1 - rd)(h)(1 - wmd) = x(1 - h)(rd).$$

These two equations define a relationship that is assumed for the simulation results that are presented here. Although these approximations are not exact, they serve as good estimates of the relative magnitude of the simulation parameters *md*, *wmd*, and *rd*.

3.3 System Parameters

The main memory cycle time is four cache cycles. It is assumed that the block is always sent in the same order, regardless of which word was referenced; the cache does not proceed until the entire block is loaded. The block size is four words, where a word is the unit of data that can be transmitted on the bus in a single cycle. The bus is held during the entire time of each bus transaction, including completion of the memory cycle if memory was accessed. Invalidation signals require one cycle of bus time; transactions involving data transfers to or from memory require the memory cycle time for the first word of the block plus one cycle for each additional word. The cache size varies from 2K to 16K words. The number of processor–cache pairs in the system varies from 1 to 15.

3.4 Simulation Parameters

For the results shown in Section 4 the following parameter values are used (see Table I). The hit ratio on private blocks varies from 95 to 98 percent. The probability that a memory reference is to a shared block ranges from 0.1 to 5 percent. The percentage of memory references that are reads varies from 85 to 70 percent. The probability that a write hit on a private block finds that block in a previously unmodified state (calculated on the basis of the equations in Section 3.2) varies from 1.75 to 5.26 percent. When a private block is selected for replacement, it is modified with respect to main memory (and hence is written back) 30–40 percent of the time. For the write-once scheme this percentage is reduced somewhat since those blocks written exactly once need not be written back. Estimates of the amount of write-backs saved vary from a few percentage points to about one-third. We include results assuming that 33 percent of the write-backs are eliminated, and also results with the pessimistic assumption that the reduction is only 5 percent. The number of shared blocks varies from 16 to 1024. The probability that a shared block reference is to the block at level i in the LRU stack is $g(1/(5 + i) - 1/(6 + i))$, where g is a normalizing factor. This

probability was selected because it results in a shared block hit ratio comparable to the private block hit ratio—slightly less since shared blocks are expected to exhibit less locality of reference than private blocks [3]. The distribution of time between successive processor requests (w of Section 3.1) is uniform from 0 to 5 cycles. Each simulation was run for 25,000 cycles. Tests indicated that extending the run time had little effect on the simulation results.

3.5 Simulation Output

Output of the simulation includes bus utilization figures, processor utilization, and a result referred to as the *system power*. This is simply the sum of the processor utilization in the system, multiplied by 100. Although a metric of effective number of processors might be more common, we use system power as the performance measure, because the uniprocessor utilization varies between coherence solutions and between simulation runs (with new parameters) of the same scheme. For example, it would be possible to have two protocols, A and B, where A is more efficient than B with a single processor, but, evaluated with ten processors, the resulting "effective number of processors" for both protocols are identical—perhaps at eight times the uniprocessor performance. Protocol A is actually more powerful and more efficient than B, but a metric using a multiple of the uniprocessor power does not reflect this difference. Defining a common uniprocessor power for all schemes and dividing the system performance of each protocol by this constant would not alter the relative position of the curves in our figures—only the labels on the vertical axis would change. Since our intent here is to determine the relative performance of the schemes, rather than the maximum number of processors possible in a particular system, we use the system power metric. As expected, the system power rises almost linearly until the bus begins to reach saturation. When a bus utilization near 100 percent is reached, the system power levels out.

4. SIMULATION RESULTS

Figures 7–16 summarize the results from four experiments chosen as representative of the simulations we have run. Each figure shows the results obtained with the indicated parameter values for all schemes from one to fifteen processors. Included in each figure are the simulation results of a simple write-through scheme in which blocks are not loaded into the cache on a write miss, and the data are written to main memory and invalidated in all other caches on each write. It should be noted that the write-through performance is somewhat inflated since it is simulated with the same hit ratio as the write-back protocols—in practice the hit ratio will be lower as a result of not loading the block into the cache on a write miss.

As was previously stated, references to shared blocks in the simulation are not necessarily references to blocks present in other caches. For a fixed number of shared blocks and a given percentage of shared block references, the *actual sharing* increases as the number of caches in the system increases. The actual sharing also varies from scheme to scheme—those approaches that invalidate other copies on a shared write have a lower level of actual sharing (as much as

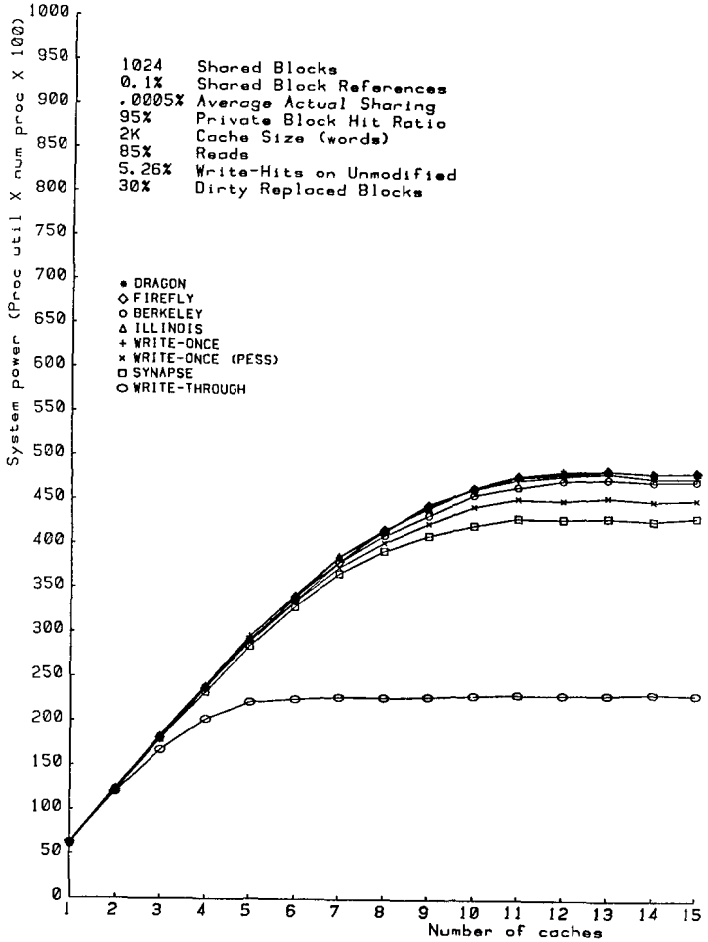

Figure 7

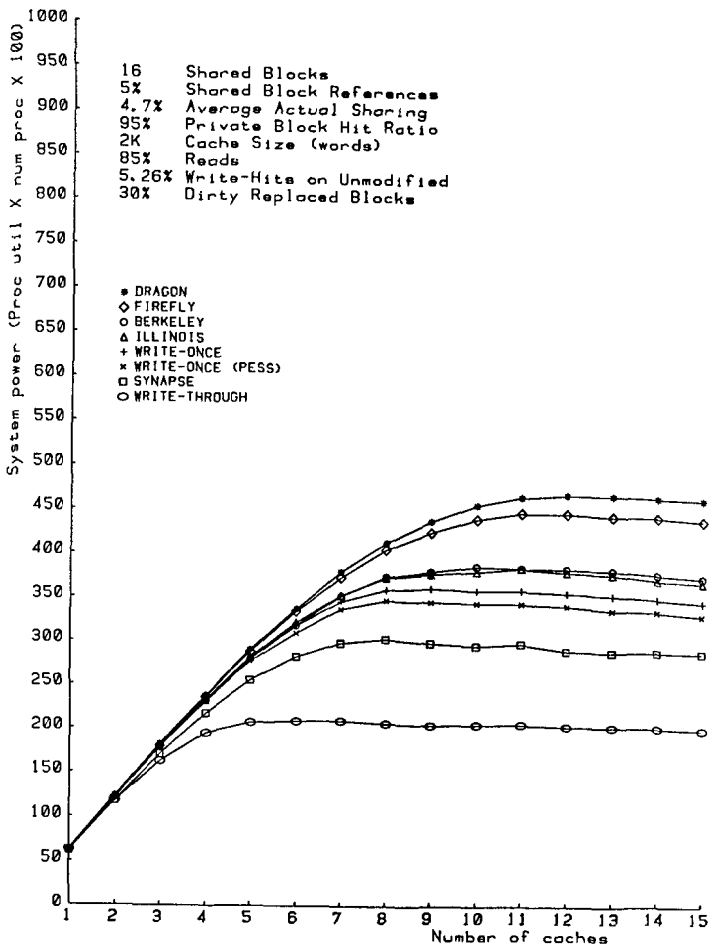

Figure 8

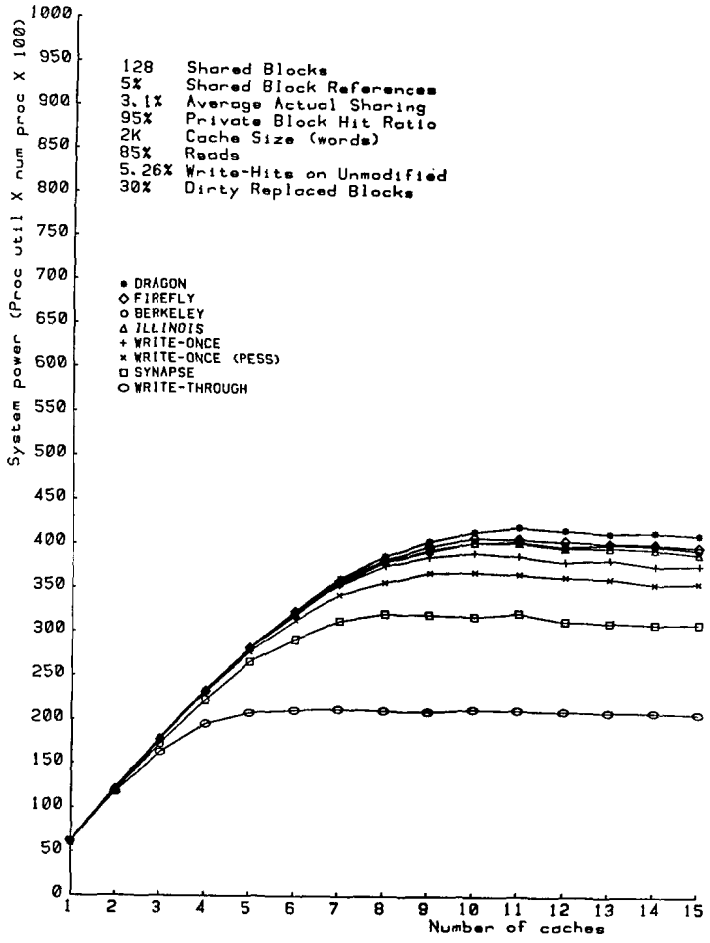

Figure 9

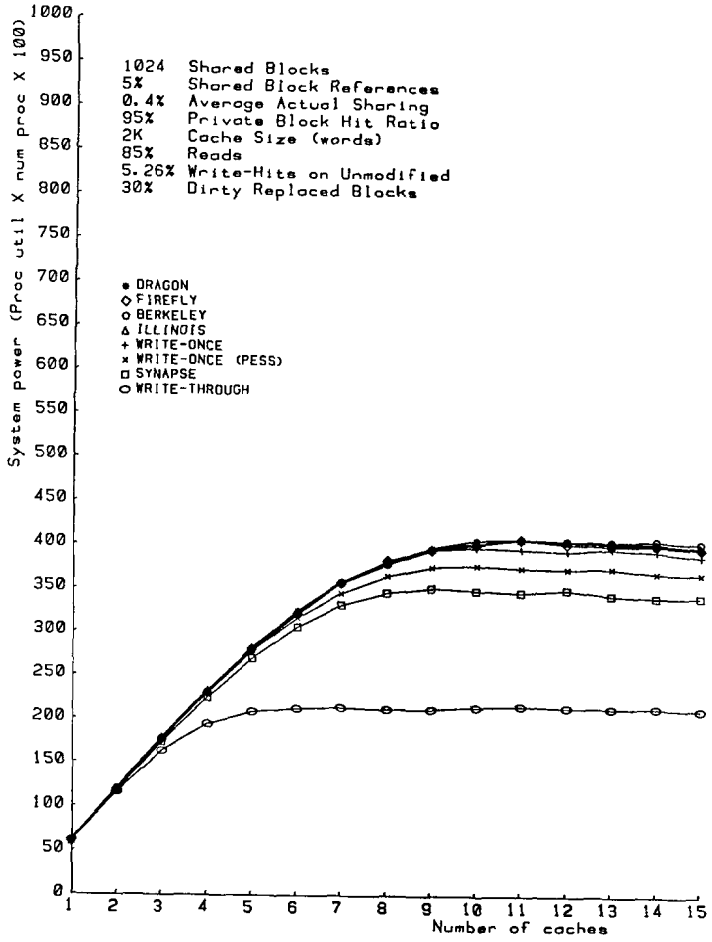

Figure 10

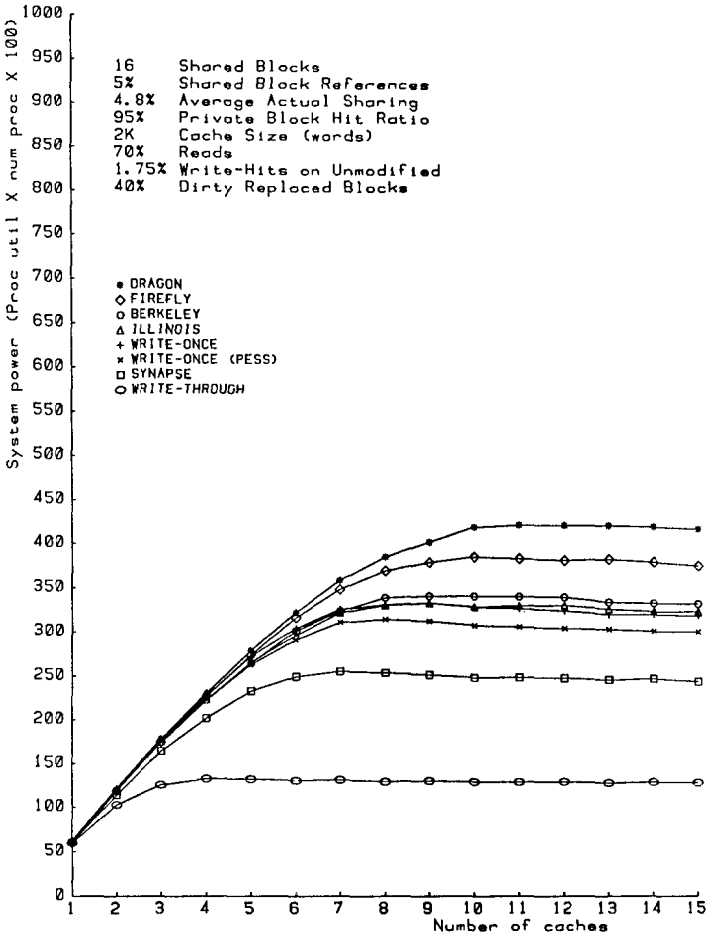

Figure 11

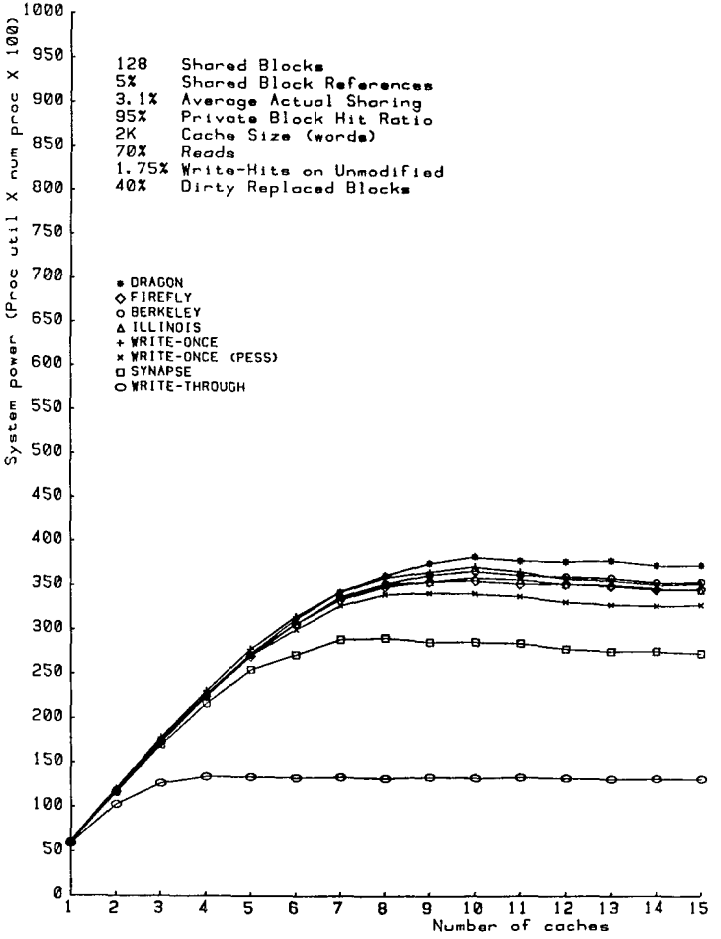

Figure 12

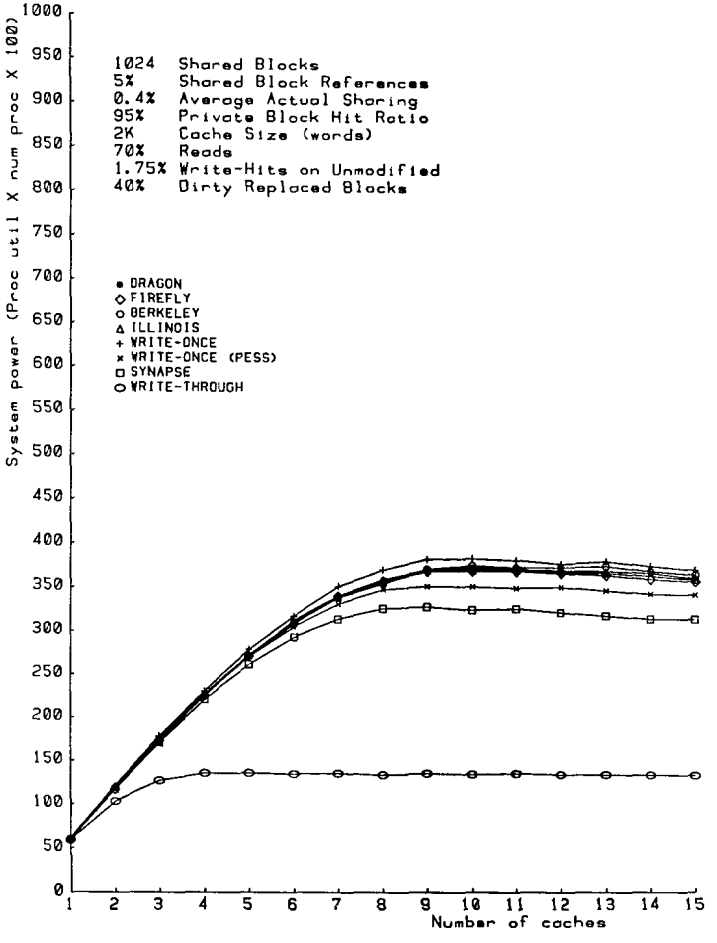

Figure 13

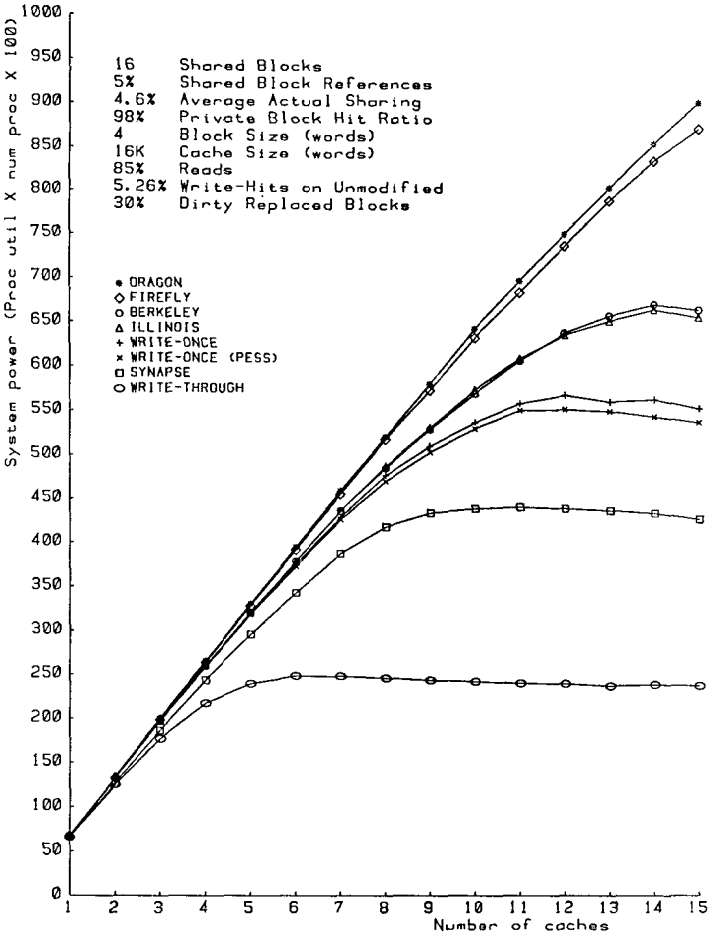

Figure 14

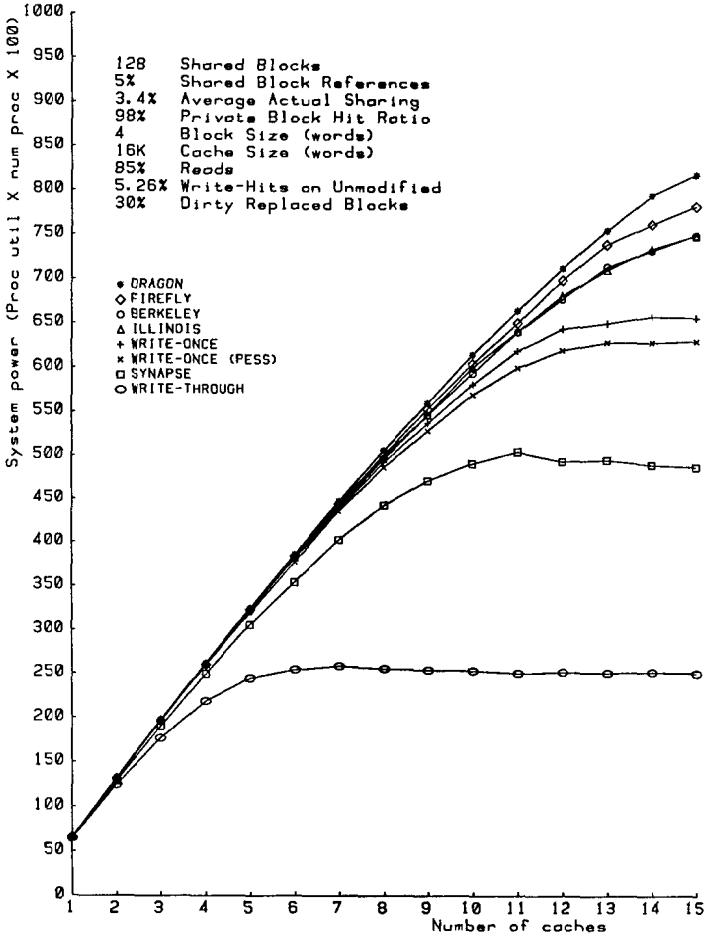

Figure 15

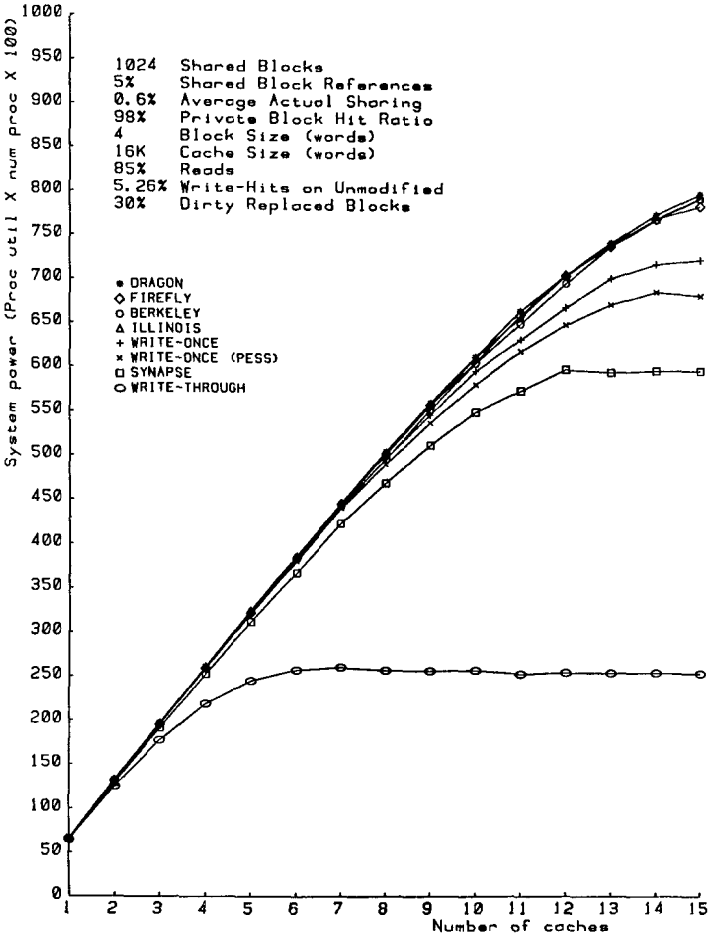

Figure 16

20 percent lower) than those approaches that distribute the new data. Each figure indicates the average level of actual sharing (the average from 2 to 15 caches), measured as the percentage of references that are to blocks present in another cache.

The first experiment (Figure 7) reflects a very low amount of sharing. (Only results with 1024 blocks are presented because results with 16 and 128 shared blocks are virtually indistinguishable.) The second experiment (Figures 8–10) reflects a higher level of shared block references. Figures 11–13 show results from a third experiment with an increased write ratio. The fourth experiment (Figures 14–16) uses a very high hit ratio and a larger cache.

4.1 Performance Issues Related to Private Blocks

As the figures demonstrate, the performance measurements of the protocols vary significantly. The first cause of dissimilar performance is efficiency in handling private blocks. Since the vast majority of all references are to private blocks, differences in their handling can be much more significant than those arising in the handling of shared blocks. Figure 7 shows the performance of the protocols with virtually no sharing. Differences between the curves are due entirely to private block overhead.

For the cache-coherence protocols that we modeled there are only two differences in the handling of private blocks: the actions that must be taken on a write hit on an unmodified block, and the actions that must be taken when a block is replaced in the cache. (The write-through method also differs in write misses and write hits on blocks that were previously written.) All schemes have identical overhead on read or write misses. The block comes from memory and requires the same transfer time. All schemes are also identical in handling read hits since the cache services the request locally and returns the data to the processor in a single cache cycle. The servicing of write hits on modified blocks requires the same time for all schemes since there is no additional overhead for a write after the first write has been completed.

In the case of write hits on unmodified private blocks the protocols span a wide range of actions. Theoretically, any overhead is logically unnecessary since private blocks are never in other caches, but only the Dragon, Firefly, and Illinois schemes are able to detect this information dynamically. In these schemes the state can be changed from VALID–EXCLUSIVE to DIRTY without any bus transaction since it is known that the unmodified block is not present elsewhere. The Berkeley scheme requires a single bus cycle for an invalidation signal. Write-once (like write-through) requires a single word write to main memory. The Synapse scheme performs a complete block load, as if a write miss had occurred.

The difference that arises in the replacement of blocks in the cache is that, for the write-once scheme, the probability that a block needs to be written back is reduced. In this scheme those blocks written exactly once are up-to-date in memory and therefore do not require a write-back.

Figure 7 indicates that the Dragon, Firefly, and Illinois schemes are identical in the handling of private blocks. Slightly below these three is Berkeley as a result of the overhead of invalidation signals. Since the signals are infrequent

(based on the parameters of this simulation) there is little degradation in performance. The performance of write-once is dependent on the trade-off between single word writes (on write hits on unmodified blocks) and the reduction in write-backs. The write to main memory is significantly more overhead in our model than an invalidation signal, but if 33 percent of the write-backs are eliminated, the overall performance equals that of the best schemes on private blocks. If, however, the reduction in write-backs is small, performance drops below that of Berkeley. The performance of the Synapse scheme is well below the others as a result of the additional overhead of treating write hits on unmodified blocks as write misses. It is important to note, however, that all protocols give much better performance than write-through.

4.2 Performance Issues Related to Shared Blocks

The remaining cause for performance differences between the protocols is overhead in the handling of shared blocks. Figures 8–16 show results with higher levels of sharing. Comparison of Figures 8–10 with Figure 7 indicates the impact of handling shared blocks efficiently—the only parameter that has changed is increased references to shared blocks. Note, for example, that the curves for Dragon, Firefly, and Illinois can be very different, although the schemes are identical with private blocks.

The protocols implement the handling of those blocks that are actually shared in a variety of ways. In fact, the only case in which they are similar is read hit. On a read miss in some schemes, the block always comes from another cache (if any cache has a copy) even if it is clean, whereas in the others the block is loaded from main memory (requiring slightly more time to service in the simulation). If the block is modified in another cache with respect to memory, some schemes require memory to be updated, but this overhead is eliminated in others with the addition of a state indicating that a block is both shared and dirty. Write misses are very similar to read misses with one major exception: Those schemes using a distributed write approach must distribute the new data to memory and/or caches after the block is loaded. Write hits on unmodified blocks require actions ranging from a complete block load to an invalidation signal. The write-back traffic also differs, since the protocols require blocks to be written back in different local states. For example, shared blocks in Dragon might be modified with respect to memory, but actual shared blocks in Firefly are always updated in main memory on every write and hence never need a write-back.

The results demonstrate that the distributed write approach of Dragon and Firefly yields the best performance in the handling of shared data. For those simulations with a small number of shared blocks (and hence more contention for those blocks) these two protocols significantly outperform the others. This is because the overhead of distributing the newly written data to everyone with a copy is much lower than repeatedly invalidating all other copies and subsequently forcing misses (followed by more invalidations) on the next references in those caches where the block was invalidated. For simulations with low contention the differences are negligible, but these protocols remain unsurpassed. Note that the performance of Dragon and Firefly actually *decreases* as the contention for shared blocks decreases and the number of shared blocks increases. In schemes with no

invalidations, the hit ratio depends only on the past references in the local cache and not on the actions of other caches. The decrease in performance is a result of an increase in the number of shared blocks in the simulation; with a larger number of shared blocks, the shared references are spread over more blocks, and there are fewer cache hits on shared blocks, since each shared block is not accessed as often. In those schemes with invalidations, the hit ratio is very dependent on the actions of other caches—for a small number of blocks the probability is high that referenced blocks are also referenced by other caches and frequently invalidated. This explains why these schemes improve as the level of contention is reduced. The performance of the Dragon exceeds that of the Firefly at levels of high sharing because the Firefly must send distributed writes to memory while the Dragon sends them to the caches only. However, this gain in performance comes at the cost of one added state (SHARED–DIRTY) for the Dragon.

The Berkeley scheme, although somewhat less efficient in handling private blocks, actually surpasses the Illinois scheme at levels of high sharing as a result of its improved efficiency in the handling of shared blocks. On a miss on a block modified in another cache, Berkeley does not require updating main memory as does Illinois. It appears that for high levels of sharing this outweighs the differences of Berkeley's invalidation signals (logically unnecessary for private blocks) and getting clean blocks from memory and not from caches as does Illinois.

The performance of write-once is lower than the above schemes (for high levels of contention) as a result of the added overhead of updating memory each time a DIRTY block is missed in another cache. In addition, the single word write to main memory (on write hits on unmodified blocks) appears to cost more than it saves in reducing write-backs of shared blocks.

The performance of Synapse is considerably lower owing to the increased overhead of read misses on blocks that are DIRTY in another cache (the originating cache must resubmit the read miss request) and to the added overhead of loading new data on a write hit on an unmodified block, as was also the case with private blocks. Synapse does, however, demonstrate significantly better performance than write-through, as do all other protocols.

4.3 Implementation Considerations

Improvements in performance are generally the result of increased hardware complexity and cost. The complexity of the bus is an important consideration. As was previously mentioned, write-once is able to work with the existing Multibus protocol without modification. The Dragon and Firefly schemes require a bus with a dedicated line to detect sharing. Similarly, the Illinois scheme assumes that a cache can detect whether a block came from memory or a cache, which could be implemented with an added bus line as with Dragon and Firefly. Both Illinois and Firefly obtain clean blocks from other caches if they are cached. In the Illinois approach, exactly one cache will succeed in putting the data on the bus—the cache with the highest priority. Since that cache may be busy servicing a memory request, the bus arbiter or prioritizer might need to wait for it to respond, increasing the service time of the request. The Firefly assumes that

all caches with a copy will succeed in putting the block on the bus, which requires a bus with fixed timing. These considerations, coupled with the possibility of slowing down the processors of those caches that succeed in putting data on the bus, have led some designers to conclude that it is more efficient to obtain the data from memory whenever possible.

All protocols but Synapse assume that each cache has the capability of inhibiting memory from responding to a request for a block when a modified copy of the block is present in that cache. The Synapse scheme uses a single bit in main memory for each block to indicate whether or not memory is to respond to requests for that block. This requires additional memory and specially designed memory controllers, but it avoids problems arising when the cache with the modified block is delayed in responding.

Additional capabilities of the bus are assumed by those schemes in which cache-to-cache transfers of modified blocks are written back to main memory at the same time (e.g., Illinois, Firefly). The added complexity of having three cooperating members on a bus can be avoided simply by performing the write-back as a separate bus operation (as with write-once) but this results in lower performance. Note that Dragon and Berkeley avoid the problem altogether since the block is not written back to memory at all (a benefit of having state SHARED–DIRTY). Synapse has no cache-to-cache transfer—the block can be loaded in the requesting cache only after the block is written back to main memory.

In our simplified model the bus remains busy until the entire memory cycle has completed, although in the case of a write, the cache is allowed to continue as soon as the data are put on the bus. One possible modification to our basic model would be to allow the bus to begin servicing the next transaction before a write has completed (assuming no contention for the same memory module). This would significantly reduce the cost of a write. Although this reduction would have little impact on relative performance in the case of write-backs, the effect on single word writes could be very significant. More precisely, the cost of a single word write could approach that of an invalidation signal, boosting the relative performance of write-once, Firefly, and write-through.

An additional issue to consider is extensions to existing protocols. Although we have only considered hardware-based protocols in this paper, it is possible to improve performance of some approaches with software assistance. For example, the Berkeley scheme includes provisions for software-based *hints* provided by a compiler or the operating system, indicating that the block is private and can therefore be loaded in the equivalent of a VALID–EXCLUSIVE state, allowing local modification without any further global interaction. This enhancement would reduce the invalidation traffic and could make the performance of the Berkeley scheme on private blocks equal to the most efficient protocols.

As was previously stated, the write-once scheme was restricted by the stipulation that it work with an existing bus protocol. A modified version of write-once has been proposed for the Futurebus [7], which would allow dynamic detection of sharing, as with Dragon, Firefly, and Illinois. In this version, blocks that are not present in other caches can be loaded in state RESERVED on a read miss, allowing modifications locally without additional overhead (just as the other

schemes use the VALID-EXCLUSIVE state). This would make write-once identical to the most efficient schemes in the handling of private blocks. Note that it would also eliminate the reduction in write-backs for private blocks, since blocks written only once would be modified with respect to main memory and would need to be written back upon replacement. Simulation results of this modified write-once scheme show overall performance to be very similar to Illinois and Berkeley.

Our simulation model assumes that the bus, cache, and processor are implemented in similar technologies and have comparable speed. On the basis of these assumptions, the maximum number of processors that can be added to a system and still result in a performance improvement ranges from about 10 (assuming a 95 percent hit ratio) to about 20 (assuming a 98 percent hit ratio). Validation of these limits is unlikely until the completion of the Dragon and Firefly workstations.

Since the bus is by far the most limiting resource, system performance can be increased considerably by increasing the capacity of the bus. This explains, at least in part, why the Synapse N + 1 can be expanded to 28 processors (some dedicated to I/O management) using two high-performance buses, and why the Sequent Balance 8000 [4] can support 12 processors using a write-through approach. The Sequent machine uses a sophisticated bus protocol, allowing the interleaving of memory requests on the bus, and it also has a second bus for synchronization purposes. In addition, in the time-sharing environment of the Sequent, the bus traffic would be reduced since a fair proportion of the 12 processors can be in an idle state at any point in time. Owing to these factors, the behavior of these two multiprocessor systems corresponds to the linear part of the performance curves—before the bus reaches saturation. Any significant increase in the number of processors would saturate the system at a lower level than would be the case with the more efficient protocols, assuming the same hardware features. The relative performance of the protocols with more efficient buses and a modified workload would remain essentially unaltered.

5. SUMMARY AND CONCLUSIONS

We have reviewed six protocols for cache coherence in shared-bus multiprocessors. Each scheme was described using a uniform terminology. A multiprocessor simulation model was presented and described, including a mechanism for simulating explicitly the dynamic reference behavior of shared data while expressing locality of references. Results using the model have been presented and discussed. The results indicate that the choice of coherence protocol in a shared-bus system is a significant design decision, since the hardware requirements vary, and since the performance differences between the protocols can be quite large. In particular, there appear to be significant differences in performance between those schemes that permit multiple writers and distribute the new data to all caches with copies and those schemes that permit only a single writer and invalidate all other copies at each write.

Among topics for future research is an investigation to determine whether there can be developed additional protocols of each type that demonstrate performance superior to the protocols described in this paper. Another interesting

topic is a study of *compatible* protocols that can be used by different caches at the same time in the same system [13]. Finally, we observe that actual run-time measurements from multiprocessors would be extremely valuable, providing more accurate parameter values and allowing validation of simulation results.

ACKNOWLEDGMENTS

We are very grateful for the extensive suggestions, comments, and corrections of Susan Eggers, Jim Goodman, and Janak Patel. We thank Ed Lazowska, Ed McCreight, Chuck Thacker, John Zahorjan, and the referees for their helpful comments.

REFERENCES

1. ARCHIBALD, J., AND BAER, J.-L. An economical solution to the cache coherence problem. In *Proceedings of the 11th International Symposium on Computer Architecture*. IEEE, New York, 1984, pp. 355–362.
2. CENSIER, L. M., AND FEAUTRIER, P. A new solution to coherence problems in multicache systems. *IEEE Trans. Comput. C-27*, 12 (Dec. 1978), 1112–1118.
3. DUBOIS, M., AND BRIGGS, F. Effects of cache coherency in multiprocessors. *IEEE Trans. Comput. C-31*, 11 (Nov. 1982), 1083–1099.
4. FIELLAND, G., AND RODGERS, D. 32-bit computer system shares load equally among up to 12 processors. *Electron. Design* (Sept. 1984), 153–168.
5. FRANK, S. J. Tightly coupled multiprocessor systems speed memory access times. *Electronics 57*, 1 (Jan. 1984), 164–169.
6. GOODMAN, J. R. Using cache memory to reduce processor–memory traffic. In *Proceedings of the 10th International Symposium on Computer Architecture*. IEEE, New York, 1983, pp. 124–131.
7. GOODMAN, J. R. Cache memory optimization to reduce processor–memory traffic. *J. VLSI Comput. Syst. 2*, 1 (1986), in press.
8. KATZ, R., EGGERS, S., WOOD, D. A., PERKINS, C., AND SHELDON, R. G. Implementing a cache consistency protocol. In *Proceedings of the 12th International Symposium on Computer Architecture*. IEEE, New York, 1985, pp. 276–283.
9. MCCREIGHT, E. The Dragon computer system: An early overview. Tech. Rep., Xerox Corp., Sept. 1984.
10. PAPAMARCOS, M., AND PATEL, J. A low overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th International Symposium on Computer Architecture*. IEEE, New York, 1984, pp. 348–354.
11. RUDOLPH, L., AND SEGALL, Z. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proceedings of the 11th International Symposium on Computer Architecture*. IEEE, New York, 1984, pp. 340–347.
12. SMITH, A. J. Cache memories. *ACM Comput. Surv. 14*, 3 (Sept. 1982), 473–530.
13. SWEAZEY, P., AND SMITH, A. J. A class of compatible cache consistency protocols and their support by the IEEE Futurebus. In *Proceedings of the 13th International Symposium on Computer Architecture*. IEEE, New York, 1986, pp. 414–423.
14. TANG, C. K. Cache system design in the tightly coupled multiprocessor system. In *Proceedings of the 1976 AFIPS National Computer Conference*. AFIPS, Reston, Va., 1976, pp. 749–753.
15. THACKER, C. Private communication, Digital Equipment Corp., July 6, 1984.
16. YEN, W. C., AND FU, K. S. Coherence problem in a multicache System. In *Proceedings of the 1982 International Conference on Parallel Processing*. IEEE, New York, 1982, pp. 332–339.

Received November 1985; revised June 1986; accepted June 1986.