Multiprocessor Cache Synchronization
Issues, Innovations, Evolution

Philip Bitar [*,**] *Alvin M. Despain* [**]

[*] Research Institute for Advanced Computer Science
NASA Ames Research Center
Moffett Field, CA 94503

[**] Computer Science Division
University of California
Berkeley, CA 94720

Abstract

Many options are possible in a cache synchronization (or consistency) scheme for a *broadcast* system. We clarify basic concepts, analyze the handling of shared data, and then describe a protocol that we are currently exploring. Finally, we analyze the evolution of options that have been proposed under *write-in* (or write-back) policy. We show how our protocol extends this evolution with new methods for efficient busy-wait locking, waiting, and unlocking. The *lock* scheme allows locking and unlocking to occur in zero time, eliminating the need for test-and-set. The scheme also integrates processor atomic read-modify-write instructions and programmer/compiler busy-wait-synchronized operations under the same mechanism. The *wait* scheme eliminates all unsuccessful retries from the bus, and allows a process to work while waiting.

Sections

A. Context

A.1. Synchronization in Shared-Memory Architecture. One of the most pressing issues in computer architecture today is how to design multiprocessor systems that can realize the speedup potential of multiple processors. In view of this, we seek to identify features of shared-memory architecture that speed up the *coordination, or synchronization,* of multiple processors.

In Project Aquarius at Berkeley, we are investigating the design of a *high-performance,* multiprocessor system to execute logic programs, currently Prolog (Dobry, Despain, Patt 1985). We seek efficient synchronization schemes that will allow us to exploit the concurrency inherent in logic programs. At the same time, we recognize that single-bus architecture offers a *low-cost* approach to building a multiple-microprocessor system, so is rapidly gaining importance (Bell et al. 1985). Our research, therefore, addresses both the high performance and the low cost approaches. A designer taking either approach will be able to select the features that best serve their interest.

A.2. Full Broadcast. We limit our attention here to multiprocessor cache systems that implement full broadcast, or more simply, *broadcast.* In such a system, at each setting of the processor-memory switch, or interconnect, every processor cache that has access to the switch can broadcast its request to all other processor sites, each of which is able to evaluate all of the requests, and service at least one of them, if appropriate. The *advantage* of this is that the operation is entirely distributed and parallel, hence is fast. In addition, main memory is simple, not needing to maintain cache state or manage cache synchronization. In this article we do not address other cache systems – partial-broadcast or non-broadcast – such as those of Tang (1976), Censier and Feautrier (1978), Yen and Fu (1982), Dubois and Briggs (1982), Archibald and Baer (1984).

For ease of discussion, we also assume that the processor-memory interconnect is a *single bus.* In fact, broadcast is currently seen only in single or dual bus systems, because this limits the number of simultaneous broadcasters to one or two, respectively, greatly simplifying implementation of broadcast.

B. Synchronization Issues

We currently identify three primary, *low-level* synchronization issues for shared-memory architecture.

- *Cache synchronization:* the sharing of replicated writable data
- *Busy wait:* locking and unlocking, waiting
- *Sleep wait:* enqueuing and dequeuing, saving and loading state

B.1. Cache Synchronization. Smith (1984) stresses the significance of the issue:

> The solution of the multicache consistency problem for large numbers of processors is one of the *most important* current problems in computer architecture, and it is one of the major barriers to effective multiprocessing.(p.19)

Specifically, processes in a shared-memory system communicate by taking *sole access* to some shared data object and writing it, leaving information there for another process to read. One example, typical of Prolog and dataflow, is the *producer/consumer* relationship. In this case, one process produces a value, say a variable binding, for another process, and that process, in turn, reads the value and uses it. The second process may report back to the first, in which case it also writes a shared-variable. Another example is the management of *service-request queues,* where one process leaves a service request for another process in the latter's request queue. The latter eventually reads the request and services it. This will typically occur among processes running on different processors. For example, a process running on a program interpreter may send a service request to a floating-point processor or an I/O processor.

The processor caches must correctly implement the read/write sharing of data that is requested by the software, across multiple sites. So the problem of *cache synchronization* consists in this:

- Read/write sharing of replicated data

B.2. Busy Wait, Sleep Wait. If the wait for sole access to a shared object is expected to be short, the process will *busy wait;* that is, it will continue to *run while waiting,* though this does not mean that it must continually test a bit while waiting. On the other hand, if the wait is expected to be long, the process will be switched out of the processor and will *sleep wait* on a queue, allowing another process to run on that processor. However, if the hardware in a multiprocessor system does not itself implement queuing, then by default the software must implement it using busy wait. In this case, a queue-manager procedure (instead of invoking hardware queues) will busy wait for access to software-implemented queues, and when it gains access to a

0884-7495/86/0000/0424$01.00 © 1986 IEEE

queue, will insert or delete a process, as appropriate. If semaphores are used, they will be part of the queue descriptor.

This identifies *the two reasons for using busy wait.*

- A situation where busy wait is *less costly* than sleep wait
- A system where busy wait is *necessary* in order to implement sleep wait

B.3. Our Focus. We address the first two issues: synchronization of caches and implementation of busy wait. Both of these issues, as well as sleep wait, are treated in Bitar (1985), to which we refer the reader who seeks further details.

C. Basic Concepts

C.1. Read/Write Synchronization. Read/write sharing of replicated data, and thus synchronizing caches, entails three logical facets: *atomicity* – sole access for writers; *concurrency* – shared access for readers; and *replication* – getting the latest version of the data upon access. But these three logical facets reduce to just two implementation requirements:

- *Serialize conflicting accesses:* write-read, write-write
- *Provide the latest version of the data:* wherever it may be

The first requirement must be met in order for the second to have meaning.

C.2. Atoms. In order for software to implement an atomic operation (insuring sole access for a writer), the hardware must provide some primitive atomic operation for the software to use. This can be as simple as writing a single bit, as in Peterson's algorithm (Peterson, Silb. 1985, p. 332). But for the sake of speed, the hardware will probably provide at least a test-and-set operation or atomic swap, allowing the software to insure sole access without so many bit reads and writes as otherwise needed.

It is crucial to recognize two types of atomic (shared, writable) data objects, then.

- *Hard atom:* Data object that is atomized by the hardware – access conflicts are serialized by the hardware.
- *Soft atom:* Data object that is atomized by the software.

An understanding of the differences is not only required to insure correctness in a cache synchronization scheme, but may also allow the design to be substantially simplified, especially for a non-broadcast system (Bitar 1985).

C.3. Cache Synchronization. The synchronization of caches, then, reduces to the following two requirements, each having the occasions shown:

- *Serialize conflicting accesses:* hard atoms only
  - *Two different processes* on two different processors access the same hard atom.
- *Provide the latest version of the data:* all writable objects
  - *Two different processes* on two different processors access the same writable, shared data (hard or soft atom).
  - *One process* on two different processors (due to migration) accesses the same writable, shared or unshared, data.

D. Shared Data

D.1. The Issue. For *unshared* data, write-in (or write-back), as opposed to write-through, has been shown to reduce bus traffic and concomitant processor idle time (Smith 1982; Norton, Abraham 1982). However, it has recently been suggested that the reverse is true for actively shared, or more simply, *shared* data. That is, when data is being shared by several caches, it may be better for a processor to write through to *update* other copies, rather than *invalidate* them (Rudolph, Segall 1984; McCreight 1984; Archibald, Baer 1985; Vernon, Holliday 1985). Accordingly, the Xerox Dragon and DEC Firefly (reported by Archibald, Baer) and Rudolph and Segall take this approach:

- *Write-through to other caches:* for actively shared data
- *Write-in:* for other, 'unshared' data

In the Dragon and Firefly protocols, a block is defined as *shared* if it currently resides in more than one cache. This status is determined when a cache writes through or fetches a block, for all caches having a copy raise the bus *hit* line, notifying the requester. In addition, on a fetch any other cache having a copy enters the block's status as *shared*. In Rudolph and Segall's scheme, a block is defined as *shared* if accesses to it are currently interleaved among the processors. Specifically, a block is unshared if a processor writes it twice (or more generally, n times, for some n) while no other processor accesses it.

D.2. Analysis. Under a *full-broadcast* system, the important distinction between write-in and write-through with regard to cache synchronization is not the policy for updating *main memory*, but the policy for updating *other caches*. In view of this, write-in and write-through may be distinguished by three features of the updates – granularity, occasion, and target.

- Write-in updates
  - *Granularity:* block
  - *Occasion:* access to block
  - *Target – demand based:* requester
- Write-through updates
  - *Granularity:* word
  - *Occasion:* write to word
  - *Target – prediction based:* all caches having a copy

Write-through actually has three potential targets – valid copies in other caches, invalid copies in other caches (discussed later), and main memory. But unless otherwise qualified, we will assume that the target consists of *valid* copies. In this context, the distinctive action on a *write to a shared block* is this:

- *Write-in: invalidate* the block in other caches
- *Write-through: update* the block in other caches

Although both policies provide the latest version of a block in the same way – from a cache or main memory, as appropriate – they serialize accesses to hard atoms differently. *Write-through* forces the processor to wait for access to the bus on every write to actively shared data. In contrast, *write-in* allows a processor to acquire the sole copy of the blocks containing an atom and write them any number of times before unlocking the atom, thereby alleviating the need to wait for bus access at every such write. In view of this, under write-in, *no other data should be placed in a block with an atom*, so that when a process locks an atom, no other process will be accessing the blocks of the atom.

We see that write-through for shared data incurs the cost of *small granularity* of updates, inappropriate for an atom whose blocks are written more than a few times while the atom is locked. Write-through also incurs the cost of updating *all* caches having copies of an atom, potentially interfering with all of the respective processors, whereas the updates that are useful are only those that update the next processor (or processors) to read the data. In effect, write-through *predicts* that if an atom is actively shared, one of the caches that has a copy will be the next to read the atom and will do so before purging its blocks. In contrast, write-in updates another cache only on *demand*.

In brief, write-through for shared data does not look so promising if the following four points are appreciated.

- A process does not access an atom until it is *unlocked* by the current user.
- Under write-in, *blocks should be devoted to atoms*, so that when an atom is locked, there is no contention for its blocks.
- Write-in is *demand* based; while write-through is *predictive*, not necessarily updating the next cache to read the atom.
- Write-through updates *all* caches having a copy of the atom, not just the next to read the atom (if at all).

The model of sharing under write-in that was introduced by Dubois and Briggs (1982) fails to appreciate the first two points, so degrades the performance of write-in.

In spite of this, one case where write-through to shared data is clearly useful is in *efficient busy wait*, where several processors may be waiting for a lock bit to be cleared, making it advantageous to broadcast writes (Section E.4).

D.3. Internal Fragmentation under Write-In. Since a block, under write-in, should be devoted to an atom that it contains, internal fragmentation of blocks can degrade performance, especially for large block size, for an entire block must be transferred when access is requested to the (possibly smaller) atom on the block. A solution is to transfer smaller *transfer units*. This can improve performance for unshared data (Goodman 1983; Hill, Smith 1984), and we can now see that it will also improve performance for atoms. To synchronize the caches, valid and dirty status must be stored with each transfer unit, and all dirty transfer units of a block must be transferred when source status (discussed later) is transferred. Alternatively, the full state can be stored with each transfer unit so that individual transfer units can be transferred on request. This appears simpler, but will require three, rather than just two, state bits per transfer unit if the protocol has more than four states for a cache block.

E. Protocol Mechanics

Before describing the evolution of broadcast protocols, we will present the mechanics of a protocol that we are currently studying. This presentation should not only to fix ideas for the reader, but will also introduce several innovations.

E.1. States. We will consider eight states for a cache block:

Invalid
Read
Read, Source, Clean
Read, Source, Dirty
Write, Source, Clean
Write, Source, Dirty
Lock, Source, Dirty
Lock, Source, Dirty, Waiter

The following is a key to the word meanings, which will be illustrated by examples in the next sections.

Invalid	Meaningless
Read	Read-only privilege (*shared-access* privilege)
Write	Read and write privilege (*sole-access* privilege)
Lock	Read and write privilege, locked by the cache
Source	Source of the latest version of the block • Location of clean/dirty status for the block • When the block is fetched by another cache, the source provides it and its clean/dirty status • When purging the block, the source flushes it if the block is dirty
Dirty	The block was written by some processor, and memory has not yet been updated
Waiter	Another processor requested the block while it was locked

E.2. Basic Actions. Figures 1-9 illustrate the interaction of the processors, caches, and memory, while Figure 10 summarizes the state transitions for a cache block. (Notice the note and template prior to Figure 1.) Keep in mind that the last cache to fetch a block becomes its *source*, and provides the block when the block is next requested by another cache (unless the source purges the block in the meantime).

Fetching Unshared Data on Read Miss. Usually a requester cache assumes *read/write/lock* privilege for a block if the request is for *read/write/lock* privilege, respectively. But Figure 1 shows that if the request is for *read* privilege and the block is *not present* in another cache – no cache signals *hit* – the requester assumes *write* privilege, so that if its processor subsequently writes the block, a bus access will not be required in order to obtain write privilege.

Fetching Without Source Cache. Figures 2 and 3 show that if there is no source cache for the block, even if the block is present in another cache, the block is provided by memory. Furthermore, the requester cache assumes *read/write* privilege for the block if the processor's request is *read/write*, respectively. But if the request is for *read* privilege, any cache that has the block signals *hit;* otherwise the requester will assume write privilege, as depicted above.

Cache-to-Cache Transfer. Figure 4 shows that if there is a *source* cache for a block, the source provides the contents of the block, if requested, along with the clean/dirty status of the block. The presence of this status on the bus signals the presence of a source cache, but the actual value is only needed if the request is for read privilege, since the block would otherwise become dirty anyway. Figure 5 shows that if the requester cache already has a valid copy at a processor write, it only requests write privilege, not the block itself.

I/O Transfer. In executing an *input* operation, an I/O processor will simply invalidate the block in all caches (requesting write privilege for the block) as it writes to memory. In executing a *paging-out* operation, the I/O processor will fetch the block for write privilege, thereby invalidating the block in all caches. In executing a *non-paging output* operation, the I/O processor will give a special read request, notifying the source cache not to give up source status. The need to distinguish between the two output operations can be avoided by flushing dirty blocks on cache-to-cache transfers (Feature 7, Section F.3). In that case, a block would simply be fetched for write-privilege by the I/O processor on all output operations.

E.3. Efficient Locking. Figure 6 illustrates how busy-wait locking can be efficiently executed in a fully associative cache. The *first block* of the atom is fetched for write privilege and locked until the entire operation is done; and the cache supplies the target word to its processor, as on a read instruction. Further, as shown in Figure 8, the unlock can occur at the final write to the block. So the *lock* instruction is a special processor *read* instruction, and the *unlock* instruction can be a special *write* instruction. An unencoded way of implementing this is to devote a separate processor line to the function, which will be interpreted by the cache as *lock* on a read and *unlock* on a write.

If another cache requests the atom while it is locked, it will request write privilege for the *first block*, as noted above, and will find it locked. The cache holding the lock will record that another cache is waiting, using the lock-waiter state (Figure 7). The requester cache, then, enters the block address in a special *busy-wait register*, setting the stage for efficient busy wait.

Efficiency. Locking a block, here, is concurrent with fetching the block, so generates no extra bus traffic, nor delays the processor. Furthermore, the first read and last write of the atom will probably be to the first block of the atom since, under the first reason for busy wait (Section B.2), the atom will probably be contained entirely on one block, and under the second reason for busy wait, the atom will probably contain the entire queue descriptor. So *locking and unlocking will usually occur in zero time.*

Therefore, cache-state locking is as fast as holding an entire cache or memory module throughout the operation, but has two important advantages.

- *Fine-grained locking:* Only the target atom is locked.
- *Trapping:* Traps do not require aborting the operation (though process-switching traps are precluded anyway, as noted below).

Compared to using a test-and-set bit, cache state locking has these advantages:

- Locking and unlocking usually occur in zero time, as opposed to fetching a lock bit and then the data.
- No blocks are devoted to lock bits (hard atoms) under write-in.

Finally, cache-state locking can be executed by a single-cycle-

instruction processor, such as a reduced-instruction-set processor (Patterson 1985). This kind of processor can now execute atomic operations as efficiently as any other.

In conclusion, cache-state locking most efficiently implements *both processor* atomic read-modify-write instructions, *and programmer/compiler*-implemented busy-wait locking.

Two Concerns. Two concerns that must be considered in this locking scheme are the need to switch processes while a block is locked, and the need to purge a locked block. In the first case, a process would be put to sleep while a block that it has locked is still in a cache. In the second case, the lock would be lost if the block must be purged to make room for another block.

With regard to *process switching*, under any locking scheme, especially busy-wait locking, it is important to preclude the switching of processes (or threads of control) while a lock is held, in order to avoid prolonging the time for which other processes may have to wait for access to the atom. This is achieved by precluding page faults, I/O, and relevant traps. Consequently, the problem should not occur. With regard to *purging a locked block*, this should not occur in a fully associative cache, due to the large set size (the entire cache), but may occur in a cache with small set size. In this case, under a minor modification to the protocol, a lock bit is written to memory when a locked block must be purged. The lock bit can be either a hardware tag bit on each memory block, or very simply, a bit, in the first block of the atom, that is reserved for hardware use by the compiler.

E.4. Efficient Busy Wait. *Purpose.* We conceive two purposes for efficient busy wait.

- Eliminate unsuccessful retries from the bus.
- Relieve a waiting processor of polling the status of a lock, allowing it to work while waiting.

In a broadcast system, the *first* purpose is achieved by broadcasting lock or unlock actions if a processor may be waiting. The *second* purpose is achieved by devoting hardware to monitoring the lock, and having it interrupt the processor when it has acquired the lock. A processor can work while waiting if it requests the lock when ready but still has work to do for a short time, executing a 'ready section' of code. We do not yet know if ready sections can actually be arranged. The next best alternative is to prefetch a lock just before it is needed, the offset depending on the expected wait time.

The primary importance of efficient waiting is to serve *the second reason for busy wait* (Section B.2). Specifically, the manipulations of the sleep-wait and ready queues that must be accessed in order for the software to implement sleep wait may require several block fetches, say three or four, per queue. And, in addition, there may be quite a few processes that access each queue, especially a global ready queue, thereby generating high contention for the queue.

Our Proposal. In our scheme, when a locked block is *unlocked*, this action is broadcast on the bus if the state in the locker cache is *lock-waiter* – indicating that another processor had requested the block while it was locked (Figure 8). A busy-wait register waiting on that lock recognizes the unlocking and joins the next bus arbitration. The winning cache will fetch the block for write privilege, lock the block using the lock-waiter state (since that will probably be appropriate), and interrupt its processor; while the other caches will let their processors continue whatever they are doing and will not access the bus, making no attempt to fetch the block again (Figure 9).

Regarding the *bus arbitration*, the waiting caches will specify very high priorities, say by using the most significant priority bit, devoted to this purpose. So if it turns out that there are no waiters after all (because the waiting processes were switched out of their processors), the arbitration will proceed normally, with no wasted time.

Basic Approaches. The two basic approaches to efficient busy wait, in contrast, derive from write-in and write-through.

- *Write-in:* When writing a lock bit, *invalidate* the corresponding block in other caches.
- *Write-through:* When writing a lock bit, *update* the corresponding block in other caches.

In either case, a waiter loops on a one in its cache (Censier, Feautrier 1978), and unnecessary invalidations or updates, respectively, are avoided by setting a lock bit only if it is zero. The lock-state protocol for locking could be modified to accommodate either of these two approaches if the cost of the busy-wait register were not warranted in the system of interest.

Rudolph, Segall (1984). Finally we observe that Rudolph and Segall (Section D.1) have oriented their cache scheme around efficient busy wait. Specifically, *write-through* is used on a processor's first write to a block after another processor has accessed the block, but *write-in* is used on subsequent writes – until another processor accesses the block. Consequently, after a lock bit is *set* (locked) on the first write to it, if another processor begins waiting before it is unlocked, the processor will read the bit, so write-through will occur again when the bit is *cleared*.

On the other hand, if no other processor begins waiting while the lock is locked, other waiters will be indirectly notified by write-in (invalidation) when the bit is *cleared* (the second write). The first waiting processor to get the bus after that will set the bit on its first write (a test-and-set). In order to have the corresponding write-through notify other waiters, whose blocks may now be invalid, write-throughs update *invalid*, as well as valid, copies. However, in order to for this to work on other data, block size is limited to *one word*. This will inflate the area of a cache devoted to addresses, and may degrade performance for other data, so we point out that one-word transfer units may be better (Section D.3). Finally, we note that the protocol also has a *block fetch* update invalid copies, but this does not seem to add any useful function since the write phase of the test-and-set will update the invalid copies anyway.

F. Protocol Evolution

Table 1 traces key steps of the evolution of broadcast, write-in protocols. The upper part shows the evolution of states, while the lower part shows the evolution of other features. The states and the features will be discussed in turn, following a glance at historical context. (The discussion of the states includes forward references to relevant features, thereby tying the two discussions together. We suggest, however, that the reader ignore the forward references on the first reading.)

F.1. Historical Context. Giving no reference to the literature, Censier and Feautrier (1978) state that the classic approach to cache synchronization, implemented in dual processor systems, is to use *identical dual directories* in each cache, devoting one directory to the processor and the other to a bus on which invalidation requests are broadcast. Write-through to main memory (not other caches) is used. But in addition to updating main memory, a write causes its address to be broadcast on the invalidation bus, and accordingly any other cache that has a valid copy of that block invalidates its entry. The cache directory monitoring the bus eliminates the interference of *irrelevant* invalidation requests – requests pertaining to a block not valid in that cache. According to Censier and Feautrier, this scheme does not, however, guarantee that conflicting single reads and writes (to hard atoms) will be serialized, for to do so would require a processor to wait for access to the bus on every write, as in write-through for shared data (Section D.2).

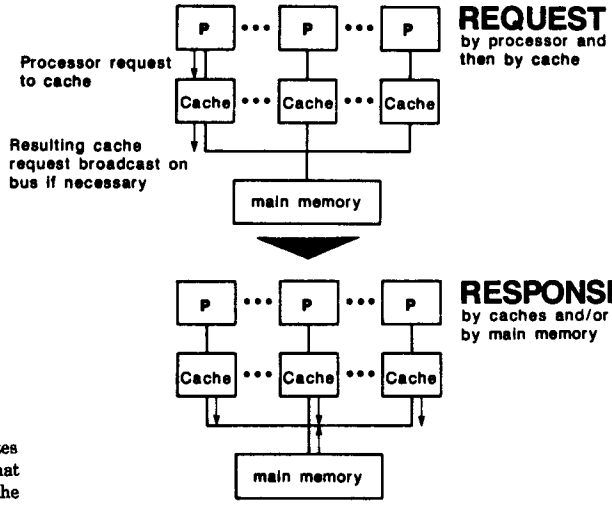
Smith (1982) and Yen et al. (1985) describe a scheme for the IBM 370/168 and 3033 that sounds similar; consequently, Censier and Feautrier may be referring to that scheme. Goodman (1983) and Frank (1984) independently reinvented the idea of

Abbreviations for Figures 1-10

B	Busy-wait register is loaded
block	Target block
C	Clean
D	Dirty
EBW	End busy wait
F	Fetch
I	Invalid
L	Lock privilege
LW	Lock waiter
O	Non-paging output operation
P	Processor
R	Read privilege
R/W	Read or write
S	Source
U	Unlock
W	Write privilege
word	Target word

Note for Figures 1-9

A state indicator inside a cache indicates that the target block is present in that cache and has that state. A blank cache indicates that the block is absent or invalid in that cache.



Template for Figures 1-9.

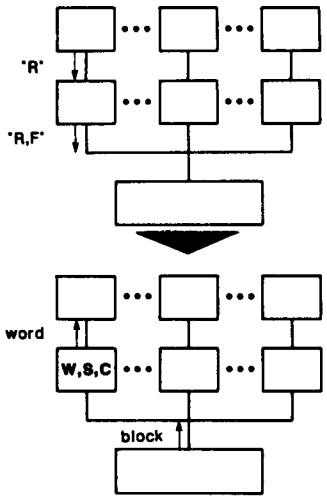


Figure 1. Fetching Unshared Data on Read Miss.

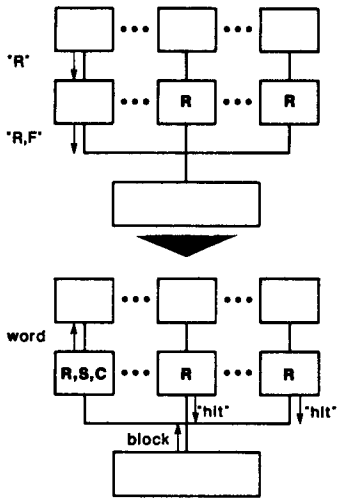


Figure 2. Fetching Without Source Cache.

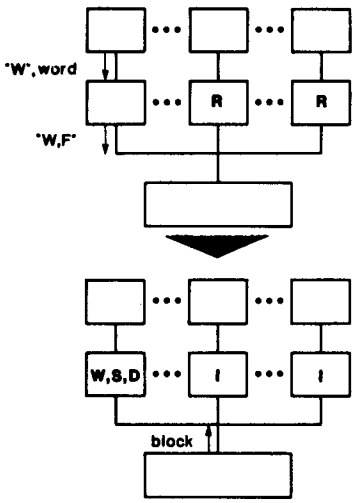


Figure 3. Fetching Without Source Cache.

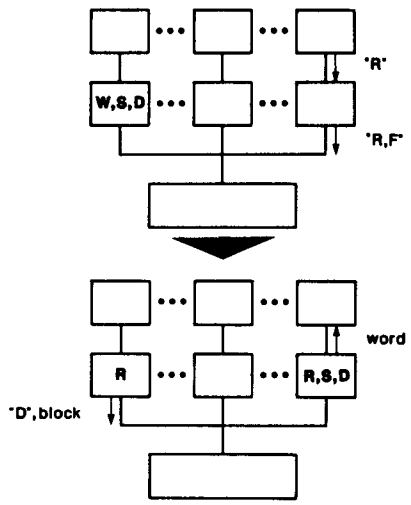


Figure 4. Cache-to-Cache Transfer.

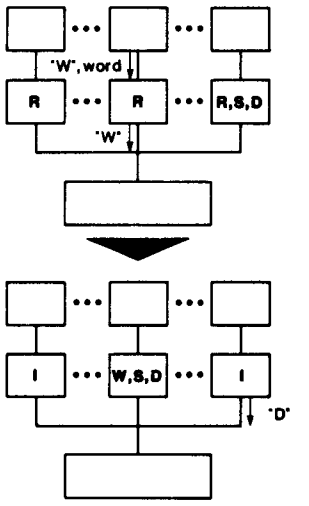


Figure 5. Request Only For Write Privilege.

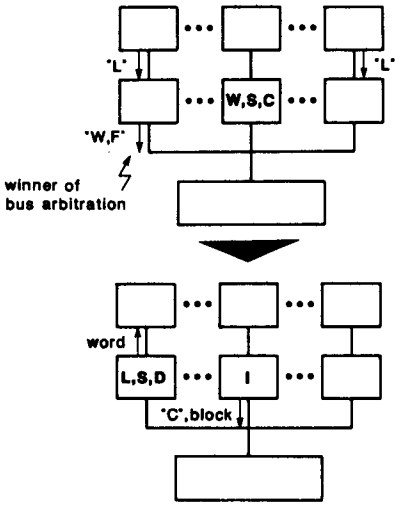


Figure 6. Locking a Block.

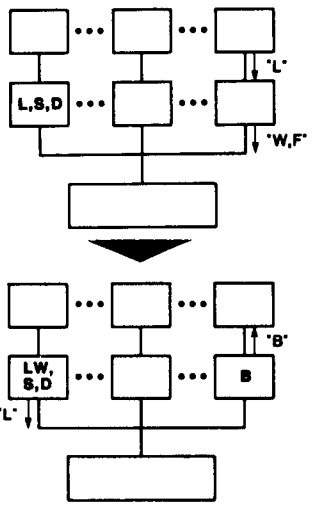


Figure 7. Requesting Locked Block; Initiating Busy Wait.

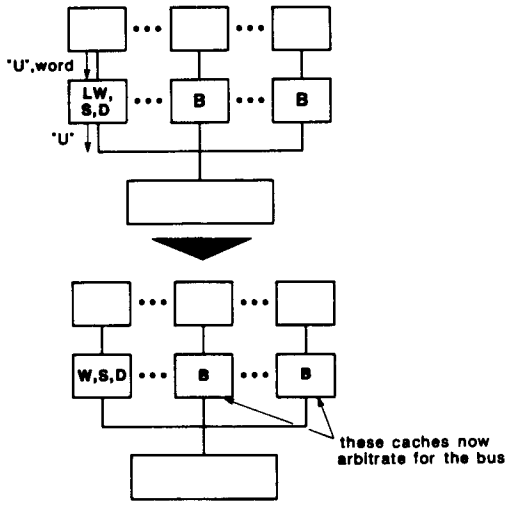


Figure 8. Unlocking a Block.

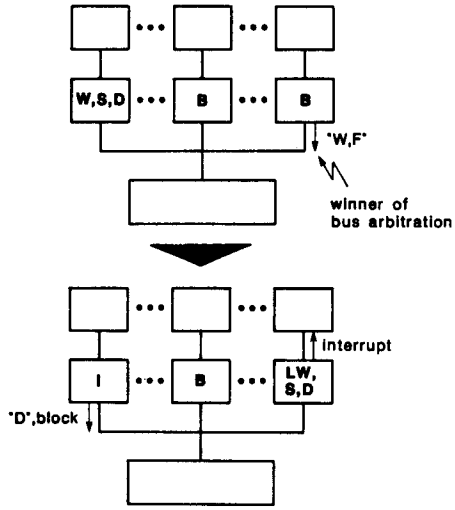
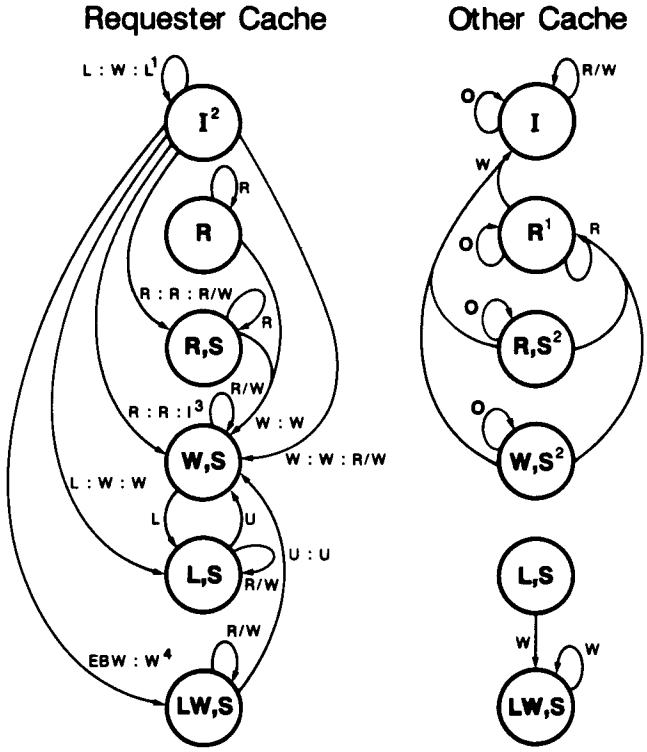


Figure 9. End Busy Wait.



Arc Label Fields: *Processor Request : Bus Request : Status in Other Cache.* (Field 3 or both 2 and 3 are omitted if irrelevant. Arcs not shown would be bugs.) **Notes:** 1. Cache then implements busy wait (Figure 7). 2. From I, bus request also fetches block (not Figure 5). 3. I in *all* other caches (Figure 1). 4. End Busy Wait (Figure 9).

Arc Label Field: *Bus Request.* (Arcs not shown would be bugs.) **Notes:** 1. Cache also signals *hit* on bus (Figure 2). 2. Cache also provides dirty status, and, if requested, the block (Figures 4,5).

Figure 10. Cache State Transitions.
(Clean/dirty status is omitted for simplicity; writing a block makes it dirty.)

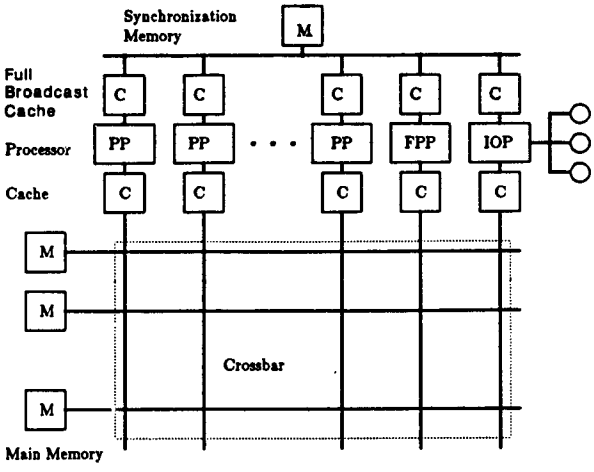


Figure 11. Aquarius Architecture.
PP = Prolog (or Program) Processor
FPP = Floating Point Processor
IOP = I/O Processor

References continued.

Peterson, J.L., Silberschatz, A. 1985. *Operating System Concepts.* Reading, Mass: Addison-Wesley, 1985.

Rudolph, L., Segall, Z. 1984. "Dynamic decentralized cache schemes for MIMD parallel processors." *11th ISCA*, 1984, 340-347.

Smith, A.J. 1982. "Cache memories." *Computing Surveys*, 14(3), Sept. 1982, 473-530. Update: Smith (1984).

Smith, A.J. 1984. "CPU Cache memories." Draft April 24, 1984. To appear in M. Flynn and G. Rossman (eds.), *Handbook for Computer Designers.* Update of Smith (1982).

Smith, A.J. 1985. "Cache evaluation and the impact of workload choice." *12th ISCA*, 1985, 64-73.

Tang, C.K. 1976. "Cache system design in the tightly coupled multiprocessor system." *NCC*, 45, 1976, 749-753.

Vernon, M.K., Holliday, M.A. 1985. "Performance analysis of multiprocessor cache consistency protocols using generalized timed petri nets." November 1985. TR 618, CS Dept., U. of Wisconsin, Madison, WI 53706

Yen, W.C., Fu, K-S. 1982. "Coherence problem in a multicache system." *Intl. Conf. on Par. Proc.*, 1982, 332-339.

Yen et al. 1985. Yen, W.C., Yen, D.W.L., Fu, K-S. "Data coherence problem in a multicache system." *IEEE-TC*, C-34(1), Jan. 1985, 56-65.

identical dual directories in the context of a write-in, bus-oriented system (Feature 3; J. Goodman, pers. comm. 1985).

F.2. States. Censier and Feautrier suggested direct *cache-to-cache transfer*, or, in our terms, the *source* function, as a useful feature for a write-in system when one cache alone has the latest version of a block requested by another cache. Goodman and Frank also independently reinvented this concept in the context of a broadcast system, applying it to the *dirty* write state.

Goodman (1983). In this scheme, a cache becomes the *source* of a block when it alone has the latest version of the block – the block is *dirty* there – which occurs in this protocol only when a cache has written the block twice. Specifically, when a dirty block is transferred from one cache to another, it is also flushed to memory, so it arrives clean. In addition, the first write to the block goes through to memory and invalidates the block in all other caches – since the original Multibus does not allow an invalidation signal while a block is fetched – so the block still remains clean. The block becomes dirty only on the second write, at which time the cache becomes the source of the block.

Frank (1984). The Synapse computer has its own proprietary bus, which enables invalidation concurrent with block fetch (Feature 4). Consequently, the *clean* write state is not useful here, as it is under Goodman's protocol, where it is the result of the invalidation write-through.

Papamarcos, Patel (1984). This scheme, however, uses the clean write state for *fetching unshared data on a read miss*, since no other process will be using the data (Figure 1; Feature 5). If the block is not written, after all, it will not need to be flushed to memory when purged. In addition, if the block is not written, the cache would not need to provide the block to another cache. However, Papamarcos and Patel do not consider the last option, for under their scheme, if a cache has a block, it also has source status for the block. This extends the *source* function from dirty to *clean* states, but is useful *only if fetching from another cache is significantly faster than fetching from memory*, since the fetch may interfere with the source's processor.

Yen, Yen, Fu (1985). The states here are those of Goodman.

Katz, Eggers, Wood, Perkins, Sheldon (1985). This scheme introduces the *dirty* read state. The write-dirty-source state is converted to read-dirty-source in a cache when another cache requests read privilege for the block. The block remains dirty because it is not flushed when is transferred. The reason for not flushing the block is that if the bus or memory does not support concurrent flushing at all, or at the speed of the caches, the flush will require an extra transfer to memory, or will slow down the cache-to-cache transfer, respectively (Feature 7).

Our Proposal. This scheme includes both *clean and dirty source* states, for the reasons already discussed, and in addition introduces the *lock* state. The lock state carries the concept of state information beyond read/write privilege to that of lock privilege, and distributes its location and control among the caches, continuing the evolutionary trend of full-broadcast cache protocols (Feature 2). The *lock-waiter* state is also proposed, to inform a cache when it must broadcast the unlocking of a block.

F.3. Features. *Feature 1.* These features were discussed above.

Feature 2: Fully-Distributed State Information. The advantage of fully distributing the state information is that it enables a cache to respond quickly to requests, the information is consolidated in just a few bits per block frame ($\lceil \log_2 \#states \rceil$), and the structure of memory is simplified. Frank, however, does not fully distribute the source status, maintaining a *source bit* in main memory, which indicates whether memory is the source or not. In contrast, *Goodman's innovation* is to fully distribute source status, and when a cache is the source, it informs memory not to provide the data when the cache services a bus request. Our proposal, in addition, distributes *lock* status.

Feature 3: Directory Duality. Goodman and Frank reestablish the classic approach of identical dual directories, and Katz et al. introduce a single, dual-ported-read directory (as well as data-store), which reduces the hardware (Borriello et al. 1985).

However, under both schemes, interference between bus accesses and processor accesses to the directory information may be generated when the processor writes to the cache, for the status of the written block must be updated to *dirty* at that time. Bus requests will be bombarding a cache continually, and according to Smith (1985), the frequency of writes may reach 35% of a processor's memory references. Therefore, one may wish to reduce, or eliminate, the interference of updating dirty status.

Two methods of *reducing* this interference are to update the dirty status only when it changes; or else in a lower performance design (Borriello et al.), to have the read and write cycles on the cache directory alternate. Another option is to eliminate the interference *entirely* by having *non-identical directories*. In this case, only the processor directory maintains clean/dirty status, for this information is accessed by the cache's bus-controller only when the cache data is accessed by the controller. Accordingly we ask, Is the frequency of *changing* a block dirty-status – the frequency of a write hit to a clean block – great enough to warrant non-identical directories? Bitar (1985) derives a formula for this frequency, then derives estimates of .2% to 1.2% from Smith's data. Thus, non-identical directories are probably not warranted on this ground.

Under our proposal, just the same, non-identical directories would also eliminate the interference of updating *lock-waiter* status by the cache's bus-controller (Section E.3), so they may still be warranted in this scheme.

Feature 4: Bus Invalidate Signal. Whereas Goodman invalidates a block by writing through to memory (Section F.2), subsequent designers assume that the bus allows explicit invalidation. Specifically, on a *write miss* the invalidate signal allows invalidating while reading the block. While on a *write hit* to a block for which the cache has only read privilege, the same signal allows a *pseudo-write (or pseudo-read)* that invalidates the block in other caches (and in Frank's protocol, clears the source bit in memory), but does not initiate a memory cycle; thus it can be limited to one bus cycle. Even so, the fractional increase in bus traffic due to the write-through is small if cache blocks are reasonably large, say n bus-wide words. This is because the increase appears to be much less than $1/n$, as estimated from Smith's data (Bitar 1985).

Feature 5: Fetching Unshared Data for Write Privilege on Read Miss. The last four protocols allow a block to be fetched for write privilege at a read miss in order to fetch unshared data. This does not reduce concurrent access to the data since the data is unshared; and if the data is subsequently written by the processor, the bus will not need to be accessed at that time in order to gain write privilege. Furthermore, the clean write state is used here, as mentioned earlier, to avoid a flush to memory if the block is not written (Papamarcos and Patel, Section F.2).

Papamarcos and Patel introduced the fetching of unshared data for write privilege by using a *dynamic* determination of sharing, namely, whether another cache currently has a valid copy or not. This uses an open collector *hit* line as do the Dragon and Firefly (Section D.1). Yen et al. and Katz et al., on the other hand, suggest a *static* determination of sharing, which is somewhat more complicated. First, the processor must have a special instruction to read data for write-privilege, which will affect a cache access only if the access is a *miss*. Second, the compiler must employ this read instruction in all reads of unshared data.

Similar to Feature 4, the fractional increase in bus traffic generated by a protocol that does not fetch unshared data for write privilege at a read miss appears to be much less than $1/n$, for

Table 1. Evolution of Full-Broadcast, Write-In (Write-Back), Cache-Synchronization Schemes

States (*Read* = *shared*-access privilege; *Write* = *sole*-access privilege)	Good. (1983)	Frank (1984)	Pap.Pat. (1984)	Yen (1985)	Katz (1985)	Our proposal
	(Regarding states: N = non-source state; S = source state)					
Invalid	N	N	N	N	N	N
Read					N	N
Read, Clean	N	N	S	N		S
Read, Dirty					S	S
Write, Clean	N		S	N	S	S
Write, Dirty	S	(S)[1]	S	S	S	S
Lock, Dirty						S
Lock, Dirty, Waiter						S
Features						
1. **Cache-to-cache transfer; serialization of conflicting single reads and writes**	√	√	√	√	√	√
2. **Fully-distributed state information:** Read / write / lock / dirty / source (R/W/L/D/S) *(faster response of caches; greater consolidation of state information; simpler memory)*	RWDS	RWD	RWDS	RWDS	RWDS	RWLDS
3. **Directory Duality:** 2 Identical Dual (ID) / 2 Non-Identical Dual (NID) / 1 Dual-Ported-Read (DPR). *(DPR reduces the hardware; NID eliminates interference due to updating status – dirty status is only in processor directory, waiter status is only in bus directory)*	ID	ID	ID[2]		DPR	NID
4. **Bus invalidate signal:** No invalidation write-through **On write hit:** Gain write privilege with a one-cycle invalidation *(instead of a word-write to memory)* **On write miss:** Gain write privilege while fetching the block *(instead of a word-write to memory)*		√	√	√	√	√
5. **Fetching unshared data for write privilege on read miss:** Unshared status is determined statically (S) or dynamically (D) *(save bus arbitration and invalidate cycle if the data is subsequently written)*			D	S	S	D
6. **Processor atomic read-modify-write instruction:** Serialize accesses		√	√		√	√
7. **Flushing on cache-to-cache transfer:** Flush block (F), or do not flush block (NF); transfer clean/dirty status with block (S) *(F is desirable unless bus and memory do not support it; NF requires transfer of clean/dirty status if source status is being transferred on a processor read and the block may be clean or dirty – see source states above)*	F	NF	F	F	NF,S	NF,S
8. **Number of sources for read-privilege block:** Allow *multiple sources*, thus a source for a read-privilege block must *always arbitrate* before providing the block (ARB); allow *loss of (single) source*, forcing the block to be *fetched from memory* (MEM); have last fetcher become source, allowing *least-recently-used replacement* across caches (LRU)			ARB		MEM	LRU, MEM
9. **Writing without fetch on write miss:** *(no fetches for process state blocks)*						√
10. **Efficient busy wait**						√

Table 1 Notes

1. A source cache provides data only for a write-privilege request, not a read-privilege request.
2. No specification is given as to whether the directories are identical or not.

blocks having n bus-wide words (Bitar 1985).

Feature 6: Processor Atomic Read-Modify-Write Instruction. There are several ways to implement processor atomic read-modify-write instructions on (hard) atoms so that accesses are serialized. We consider four methods. Only the first requires going through to memory, but the first three do require that the processor inform the cache, at the start of the instruction, that the instruction is an atomic read-modify-write.

The *first* method requires a read-modify-write instruction to *access and hold the main memory unit* that contains the target atom, throughout the operation (Rudolph, Segall 1984).

The *second* method, apparently that of Frank, requires that the atom be contained entirely on one block (appropriate for write-in, anyway), that the block be fetched for sole-access (write) privilege at the *beginning* of the read-modify-write instruction, and that the cache (or cache module) be held throughout the operation. Papamarcos and Patel propose a variant: if the cache does not have write privilege for the block at the beginning of the operation, *the bus is gotten and held through to the write*, at which time write privilege for the block is obtained as usual. We do not see an advantage in this special case, over that of fetching the block for write privilege at the beginning of the operation, while the disadvantage is that the bus is held longer than needed. We also point out that a processor read instruction that is used for fetching unshared data for write privilege (Yen et al., Katz et al.) will not in general work here, since it only applies on misses. Just the same, Katz et al. are actually planning to have their *cache*, rather than their processor, execute test-and-set in a manner similar to that depicted for Frank's processor, and they are not concerned with implementing other atomic read-modify-write instructions (R. Katz, personal communication 1986).

Under the *third* method, the cache does not fetch the block for write privilege until the write, nor does it hold the bus in the meantime. So if the write generates a *miss*, it means that the block was stolen between the read and the write, and atomicity is violated. Thus the cache raises an *exception* that causes the processor to abort the instruction, and the cache aborts the pending write request.

The *fourth* method is to use the cache lock-state to lock just the target atom (Section E.3).

Feature 7: Flushing on Cache-to-Cache Transfer. When transferring a block from one cache to another, there are three advantages to flushing it.

- If the block is dirty: *Reliability* in the face of subsequent cache failures is increased.
- If source status is being transferred on a processor read and the block may be clean or dirty: *Clean/dirty status* need not be transferred.
- In our protocol, only one output operation is needed (Sec. E.2).

Keep in mind that a protocol supports cache-to-cache transfer only from a cache having source status for the block (indicated at the top of the table).

In view of this, if a source can have *either clean or dirty status* and source status can be transferred on a processor read, (Papamarcos and Patel, Katz et al., our proposal), then clean/dirty status should be transferred along with the block, unless the block is flushed to memory while transferred – as it is in the Papamarcos and Patel scheme. Papamarcos and Patel, just the same, flush only dirty blocks, so clean/dirty status must, in effect, be put on the bus in their protocol, anyway. If memory can keep up with the flushes and if available bus codes are scarce, it may be useful to flush *all* blocks so that two different codes are not needed for cache-to-cache transfer.

Due to its advantages, flushing should be implemented if it can be done concurrently with the transfer at the speed of the caches. Even so, we depict the non-flush option in order to elucidate the more complex option that will be necessary under many buses. We also point out that the need to transfer clean/dirty status in the Katz et al. protocol can be eliminated by giving their clean write state non-source status. (This state is entered only on a read miss to unshared data.) This eliminates an inconsistency in the protocol as well, namely, giving the clean write state source status, but not doing the same for a clean read state. For the reason for a clean source state is that fetching from another cache is significantly faster than fetching from memory (Papamarcos and Patel, Section F.2).

Feature 8: Number of Sources for Read-Privilege Block. Under Papamarcos and Patel, if a block is in any cache, it is fetched from a cache, rather than from memory. Yet if the block has *read* status, several caches may have the block, so any such cache must arbitrate in order to select the actual source. This is done so that only one cache may interfere with its processor, and if necessary, to limit the number of devices driving the bus. *Arbitration slows down the cache-to-cache transfer*, however, increasing the bus traffic, as well as the requester wait.

Under Katz et al. and our proposal, in contrast, arbitration of potential sources is never required. Yet, if a block has read status in several caches and the source purges the block (flushing it to memory if dirty), there will be *no source cache* for the block. So the next fetch of the block must be serviced by memory (Figures 2,3), a disadvantage if a fetch from memory is slower than cache-to-cache transfer with arbitration. If *LRU replacement* tends to hold across caches, however, our protocol can take advantage of it since the last cache to fetch a block always becomes the new source, reducing the chance of losing a source.

Feature 9: Writing without Fetch on Write Miss. Under write-without-fetch, if the processor is going to write all of the data in a block, the block need not be fetched on a miss, though the bus must be accessed in order to invalidate the block in other caches, as usual. In order to implement this, the compiler must know when a processor will write all of the data in a block. This may occur in initializing data, but more importantly, in *saving state at a process switch*. In the Aquarius system, for example, we anticipate frequent process switching, hence the switching must be very efficient. The processor must also have a way to inform the cache of this kind of write.

Feature 10: Efficient Busy Wait. Among the protocols shown in the table, only ours makes it clear how efficient busy wait can be achieved (Section E.4).

Feature 11: I/O Transfer. Although not itemized in the table, a protocol must explicate how I/O is performed (Section E.2).

G. Conclusion

G.1. Feature Evaluation. The innovations that have been described are shown in Table 2. The extent to which any feature improves performance needs to be evaluated for the particular system of interest. The system of immediate interest to us is the Aquarius multiprocessor Prolog architecture, whose design is being developed (Dobry, Despain, Patt 1985). Figure 11 shows the two switch-memory systems of the architecture. The upper one, having a single bus, contains the program *synchronization data*, while the lower one, having a crossbar, handles *instructions and non-synchronization data.*

A separate switch-memory system for synchronization was proposed because we intend to implement Prolog predicates (procedures) as lightweight processes, thereby generating many medium-grained, lightweight processes and many synchronization operations in the system. Consequently, the speed advantage of full broadcast will be of great value. But in order to avoid the high cost of implementing full broadcast in a high-concurrency switch, such as a crossbar, we will implement broadcast in a separate system using a single bus, as shown. The

caches in that system will follow a full-broadcast synchronization protocol, and the options presented in this article will be evaluated as to their effect on performance in this cache system. Furthermore, all hard atoms will reside in the upper system, thereby simplifying the lower cache system. In particular, the latter will not need to serialize accesses to a block, but will only need to provide the latest version of each block.

G.2. Overview. We have seen that many options for broadcast cache-synchronization schemes have been proposed since Goodman's paper in 1983. We have further seen that a cache can play a crucial role in efficient busy-wait locking and waiting, and we believe that our proposals of the lock state and busy-wait register are promising, especially in the *synchronization system of the Aquarius architecture*. In this system, an improvement in the efficiency of busy-wait locking and waiting may offer a significant improvement in performance since the resulting traffic will constitute a relatively large fraction of the whole in that system. Finally, we look forward to obtaining performance statistics for our system, as well as availing ourselves of the much-needed work of others in this direction (e.g., Papamarcos, Patel 1984; Archibald, Baer 1985; Vernon, Holliday 1985).

Acknowledgements

The first author is grateful to Peter Denning for providing the opportunity to pursue this research at RIACS, summer 1985. We thank Yale Patt, Jim Goodman, Vason Srini, Steve Melvin, and George Adams for valuable insights, stimulating ideas, and helpful criticism. We also appreciate interactions with Alan Smith, Randy Katz, Mark Hill, David Wood, Susan Eggers, Jim Archibald, Mike Karels, and Chien Chen, as well as the support of the other members of RIACS, the Aquarius team at Berkeley, and the U.C. Computer Science Division staff.

This work was supported by RIACS grant NAS 2-11530, and DARPA (DoD) order 4871, monitored by Naval Electronics Systems Command under contract N00039-84-C-0089.

References

Several Abbreviations:

CS	Computer Science
IEEE-TC	IEEE Transactions on Computers
ISCA	International Symposium on Computer Architecture
NCC	AFIPS Conf. Proc., National Computer Conference
TR	Technical Report

Archibald, J., Baer, J-L. 1984. "An economical solution to the cache coherence problem." *11th ISCA*, 1984, 355-362.

Archibald, J., Baer, J-L. 1985. "An evaluation of cache coherence solutions in shared-bus multiprocessors." Oct. 1985. TR 85-10-05, CS Dept., U. of Washington, Seattle, WA 98195.

Bell et al 1985. Bell, C.G., Burkhart, H.B. III, Emmerich, S., Anzelmo, A., Moore, R., Schanin, D., Nassi, I., Rupp, C. "The Encore continuum." *NCC*, 54, 1985, 147-155.

Bitar, P. 1985. "Fast synchronization for shared-memory multiprocessors." Dec. 1985. TR 85.11, Research Institute for Advanced Computer Science, NASA Ames Research Center, MS 230-5, Moffett Field, CA 94503. Several errors in the TR have been corrected above.

Borriello et al. 1985. Borriello, G., Eggers, S., Katz, R., McKinley, H., Perkins, C., Scott, W., Sheldon, R., Whalen, S., Wood, D. "Design and implementation of an integrated snooping data cache." Jan. 1985. TR UCB/CSD 84/199, CS Division, U. of California, Berkeley, CA 94720. Sequel: Katz et al. (1985).

Censier, L.M., Feautrier, P. 1978. "A new solution to coherence problems in multicache systems." *IEEE-TC*, C-27(12), Dec. 1978, 1112-8.

Dobry, T.P., Despain, A.M., Patt, Y.N. 1985. "Performance studies of a Prolog machine architecture." *12th ISCA*, 1985.

Dubois, M., Briggs, F.A. 1982. "Effects of cache coherency in multiprocessors." *IEEE-TC*, C-31(11), Nov. 1982, 1083-1099.

Frank, S. 1984. "Tightly coupled multiprocessor system speeds memory-access times." *Electronics*, Jan. 12, 1984. Description of the Synapse computer system.

Goodman, J.R. 1983. "Using cache memory to reduce processor-memory traffic." *10th ISCA*, 1983, 124-131. Update: TR 580, CS Dept., U. of Wisconsin, Madison, WI 53706.

Hill, M.D., Smith A.J. 1984. "Experimental evaluation of on-chip microprocessor cache memories." *11th ISCA*, 1984, 158-166.

Table 2. Innovation Summary

Early Schemes (Sections F.1, F.2, E.4)

- **Classic (pre-1978) – write-through**
  - ☐ Identical dual directories
  - ☐ Broadcast an invalidation request on every write
- **Censier, Feautrier (1978) – partial-broadcast, write-in**
  - ☐ Cache-to-cache transfer for *dirty* blocks
  - ☐ Primitive efficient busy wait – loop on block in cache

Full Broadcast, Write-In (Sections F, E.3, E.4)

- **Goodman (1983)**
  - ☐ Identical dual directories
  - ☐ Fully-distributed read/write/dirty/source status
  - ☐ Cache-to-cache transfer (source status) for *dirty* blocks
  - ☐ Flushing on cache-to-cache transfer
  - ☐ Serializing conflicting single reads and writes
- **Frank (1984)**
  - ☐ Bus invalidate signal
  - ☐ No flushing on cache-to-cache transfer
- **Papamarcos, Patel (1984)**
  - ☐ Cache-to-cache transfer (source status) for *clean* blocks
  - ☐ Fetching unshared data for write privilege on read miss – *dynamic* determination of unshared status using *bus hit line*
  - ☐ Multiple sources for read-shared block; a read-privilege source *arbitrates* before providing a block
  - ☐ Serializing atomic read-modify-writes
- **Yen, Yen, Fu (1985)**
  - ☐ Fetching unshared data for write privilege – *static* determination of unshared status using *program declaration*
- **Katz, Eggers, Wood, Perkins, Sheldon (1985)**
  - ☐ Cache-to-cache transfer for read request, without flushing – *dirty* read state
  - ☐ *Dual-ported-read* directory and data-store
  - ☐ Single source for read-shared (dirty) block – *fetch from memory* if source purges block
- **Our proposal**
  - ☐ Efficient busy-wait locking – *lock* state
  - ☐ Efficient busy-waiting – *lock-waiter* state, *busy-wait register*
  - ☐ Analysis of interdirectory interference
  - ☐ Single source for read-shared block, but *last fetcher becomes source*, allowing LRU replacement across caches
  - ☐ Writing without fetch on write miss, to save process state

Write-In/Write-Through Schemes (Sections D.1, E.4)

- **Write-in** for unshared data, **write-through** for shared data
- **Dragon, Firefly (McCreight 1984; Archibald, Baer 1985)**
  - ☐ *Dynamic* determination of shared status using *bus hit line*
- **Rudolph, Segall (1984)**
  - ☐ *Dynamic* determination of shared status using *interleaving of accesses* among the processors
  - ☐ Efficient busy wait

Katz et al. 1985. Katz, R.H., Eggers, S.J., Wood, D.A., Perkins, C.L., Sheldon, R.G. "Implementing a cache consistency protocol." *12th ISCA*, 1985, 276-283.

McCreight, E.M. 1984. "The Dragon computer system." NATO Advanced Study Institute on Microarchitecture of VLSI Computers. Urbino, Italy, 1984.

Norton, R.L., Abraham, J.A. 1982. "Using write back cache to improve performance of multiuser multiprocessors." *Intl. Conf. on Par. Proc.*, 1982, 326-331.

Papamarcos, M.S., Patel, J.H. 1984. "A low-overhead coherence solution for multiprocessors with private cache memories." *11th ISCA*, 1984, 348-354.

Patterson, D.A. 1985. "Reduced Instruction Set Computers," *CACM*, 28(1), Jan. 1985, *8-21*.

References are continued following Figure 9.