IMPLEMENTING A CACHE CONSISTENCY PROTOCOL[1]

R. H. Katz, S. J. Eggers, D. A. Wood., C. L. Perkins, R. G. Sheldon

Computer Science Division
Electrical Engineering and Computer Science Department
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT: We present an *ownership-based* multiprocessor cache consistency protocol, designed for implementation by a single chip VLSI cache controller. The protocol and its VLSI realization are described in some detail, to emphasize the important implementation issues, in particular, the controller critical sections and the inter- and intra-cache interlocks needed to maintain cache consistency. The design has been carried through to layout in a P-Well CMOS technology.

Key Words and Phrases: Multiprocessor Cache Consistency, Single Chip Implementation, Snooping Caches, Ownership-Based Protocols.

1. Introduction

Advances in integrated circuit technology make it possible to achieve ever denser levels of integration. Because of pin bandwidth limitations, one effective way for VLSI processor architects to take advantage of the increasing number of transistors is to place more memory on-chip as instruction and data caches. The next generation of commercial microprocessors already dedicate some of their transistor resources to on-chip instruction memories (e.g., Motorola 68020).

Shared-memory multiprocessor systems, with caches associated with each processor, require a mechanism for maintaining *cache consistency;* i.e., all cache entries for the same block of memory must have identical values. For example, if two processors locally cache the same memory location, and one updates the location without informing the other, then an inconsistent state will arise. Two processors reading the same location will obtain different results! The cache controller must participate in the implementation of a *cache consistency protocol* for maintaining a consistent view of memory across the caches. The protocols must be designed carefully if good system performance is to be achieved.

---

[1]Research supported by Defense Advanced Projects Agency (DoD) under contract N00034-K-0251.

There has been a great deal of recent interest in cache consistency protocols (e.g., [ARCH84, GOOD83, MCCR84, PAPA84, RUDO84]). In this paper, we compare several protocols and propose one which is an improvement on these. It has been designed for implementation by an *integrated snooping data cache,* a single chip system consisting of (1) a data cache memory, (2) a cache controller that interfaces with the processor, and (3) a "snooping controller" that monitors the system bus. The cache and snoop controllers together implement the protocol.

In section 2, the multiprocessor cache consistency problem is described. We present our cache consistency protocol, and compare it with other proposals. Section 3 highlights the aspects of the chip architecture that are relevant to the protocol implementation. The implementation is analyzed for its critical sections in section 4, where we argue that the implementation is race-free. Our summary and conclusions are given in section 5.

2. Multiprocessor Cache Consistency

2.1. Consistency Issues and Protocols

From the programmer's viewpoint, a system with a cache behaves functionally as one without a cache. Therefore, when caches are distributed among multiple processors, the system must ensure that a consistent view of memory is maintained. A read of a block by one processor, following its write by another, must obtain the updated block. One solution is to make uncacheable those portions of memory that are write shareable. The alternative is to implement a *cache consistency protocol.*

In this paper, we focus on protocols for *Snooping Data Caches.* The Snoop monitors system bus transactions and may manipulate the cache based on these *external* requests. Its sophistication varies with the protocol it implements.

Three kinds of snooping cache protocols have been proposed. A *write-through* strategy [AGRA77] writes all cache updates through to the memory system. Caches of other processors on the bus must monitor bus transactions, and invalidate any entries that match when the memory

block is written through. A processor's performance is significantly degraded on writes, because of the additional latency of writing to main memory.

A second strategy is called *write-first* [GOOD83, THAC82]. On the first write to a cached entry, the update is written through to memory. This forces other caches to invalidate a matching entry, thus guaranteeing the writing processor that it holds the only cached copy. Subsequent writes can be performed in the cache. A processor read will be serviced either by the memory or by a cache, whichever has the most up-to-date version of the block. This protocol is more complicated to implement than write-through, because the Snoop must service external read requests as well as invalidate cache entries. A potential disadvantage of write-first is that it incurs an initial write to memory even when a memory block is not being shared. However, this represents an extra memory write only if there are further processor writes to the memory block.

The third strategy is called *ownership* (e.g., [FRAN84]). A processor must "own" a block of memory before it is allowed to update it. Ownership is acquired through special read and write operations. By thus predeclaring an intention to update a portion of memory, the "invalidating" writes exhibited in the above protocols are avoided. However, additional bus transactions may be incurred if the processor does not correctly predeclare its intentions.

A new class of protocols are emerging based on *write-broadcast* [RUDO84, MCCR84]. In these, a Snoop responds to an external write, not by invalidating a matching cache entry, but by overwriting it with the new value. For example, [RUDO84] describes a protocol similar to write-first, except that the first write is broadcast to other caches while being written through to memory. A subsequent write by the same cache generates a bus operation to invalidate other copies. Thus, the caches dynamically distinguish between local blocks (those to whom multiple writes are directed by the same processor) and shared blocks (those to whom local writes are interleaved with external reads and writes). The protocol improves on write-first for shared variables, but makes multiple writes to non-shared variables even more expensive.

Some protocol issues are independent of the strategy employed. These are: (1) whether the memory controller is "dumb" or "smart": either memory is inhibited by the responding cache or the memory controller knows when it does not own the requested block, and (2) whether shared blocks are returned to global memory or kept in the caches: e.g., both [GOOD83] (write-first) and [FRAN84] (ownership) require that the cache return a block to main memory after it has been accessed by another processor. The Berkeley Protocol, described below, implements an ownership strategy with owning caches inhibiting memory and owned blocks being kept in the cache.

2.2. The Berkeley Ownership Protocol: Concepts and Examples

First we define some terms. A *block* is a logical unit of memory consisting of one or a small number of words. It is identified by its address, and is the unit of transfer between main memory and the caches. Copies of a block's contents can simultaneously reside in main memory and/or in several of the cache memories.

A *cache entry* is a physical slot within cache memory that consists of a data portion, a tag, and a state. It is analogous to a page frame in a virtual memory system. The *data portion* holds the cached copy of a memory block. The *tag* is the portion of the block address that is used by cache's address mapping algorithm to determine whether the block is in the cache. Since different blocks are mapped to the same entry, the tag distinguishes among these. The *state* encodes the state of the data portion of the cache entry. For the Berkeley Protocol, the possible states are: **Invalid, UnOwned, Owned Exclusively,** or **Owned NonExclusively.**[2]

Copies of a memory block can reside in more than one cache. At most one cache *owns* the block, and the owner is the only cache allowed to update it. Owning a block also obligates the owner to provide the data to other requesting caches and to update main memory when the block is flushed from the cache. If the state is **Owned Exclusively,** the owning cache holds the only cached copy of the block. Updates can occur without informing the other caches. A state of

[2] **States** are shown in boldface, while *Bus Operations* are shown in italics. In addition, new concepts are introduced in italics.

Owned NonExclusively implies that other caches have copies and must be informed about updates to the block. The **UnOwned** state carries neither ownership rights nor responsibilities for a cache. In this case, several caches may have copies of the block. A brief summary of the implications of each state are given in Table 2.1.

A cache entry's state can change in response to a system bus or processor operation that affects its validity or exclusiveness. The system bus supports conventional *Read* and *Write* operations, as well as additional protocol specific operations (see Table 2.2). A more complete description of each operation will be given in section 2.3.

Table 2.1 -- Summary of Cache Entry States

Abbrev	State	Description
INV	Invalid	does not contain useful data.
UNO	UnOwned	contains a valid block, possibly shared among other caches; cannot be written locally without acquiring ownership first.
EXC	Owned Exclusively	the entry's block is unique, therefore data can be updated locally; its data must be given to any requesting cache and (eventually) flushed back to main memory.
NON	Owned NonExclusively	the entry's block is owned, but it cannot be updated without informing the other caches.

Table 2.2 -- Bus Operation Summary

Operation	Description
Read	a conventional read, gives a cache an UnOwned copy of the block; the data may be provided by a cache owner rather than by main memory.
Write	a conventional write, causes main memory to be updated and all cached copies to be invalidated; only issued by I/O devices and other bus users without caches.
Read-For-Ownership	like a normal read except that the requesting cache becomes the exclusive owner after the read completes.
Write-For-Invalidation	a quick version of the conventional write; need not entirely update main memory, but is guaranteed to invalidate any other cached copies; main memory will be updated correctly later when the (owned) block is flushed from its cache.
Write-Without-Invalidation	main memory is updated, but any cached copies are kept valid; used for flushing owned blocks to memory, but not necessary for correct operation since a normal Write could be used; having the extra operation avoids unnecessary invalidations.

We can now present some examples to show how the protocol behaves for different processor requests in various states. Each of the following four figures shows the states of different caches containing copies of the same block both before and after the operation. We adopt a three letter abbreviation to indicate the state of the entry: **INV** for **Invalid**, **UNO** for **UnOwned**, **EXC** for **Owned Exclusively**, and **NON** for **Owned NonExclusively.** All entries with an "A" next to their state contain the same data. A dotted line indicates that data is being sent along the system bus.

Figure 2.1 shows a *Read* being serviced by main memory. The Cache Controller of the requesting processor (Cache N) determines that its entry is **Invalid.** It generates a bus *Read*, which is handled by main memory in the usual fashion. The new entry is marked **UnOwned,** since it now contains a valid but read-only copy of the block. Notice that another cache already had a copy of the block. Since that cache did not own the block, it did not respond to the request.

Figure 2.2 depicts a *Read* in which the block is supplied by another cache rather than main memory. The Cache Controller of Cache N determines that its block is **Invalid** and generates a bus *Read.* Cache 1's Snoop Controller detects that the read is for a block that it owns. It inhibits main memory and sends the data over the system bus. Its Snoop then changes the entry's state to **Owned NonExclusively,** because now another cache has a copy of the block. Cache N marks its copy **UnOwned.** If additional requests are made for the same block, the owning Snoop in Cache 1 will continue to supply the data but will not need to change its local state again -- it remains **Owned NonExclusively.**

Figure 2.3 shows a *Write* being handled by the Cache N's Cache Controller. Since that cache already has a valid copy of the block, it does not need to generate a *Read-For-Ownership* to acquire the block. It can "steal" ownership by issuing a *Write-For-Invalidation* and then updating its copy locally. The Snoops in the other caches invalidate their local copies of the data. Cache N marks its newly updated entry as **Owned Exclusively,** since it now has the only

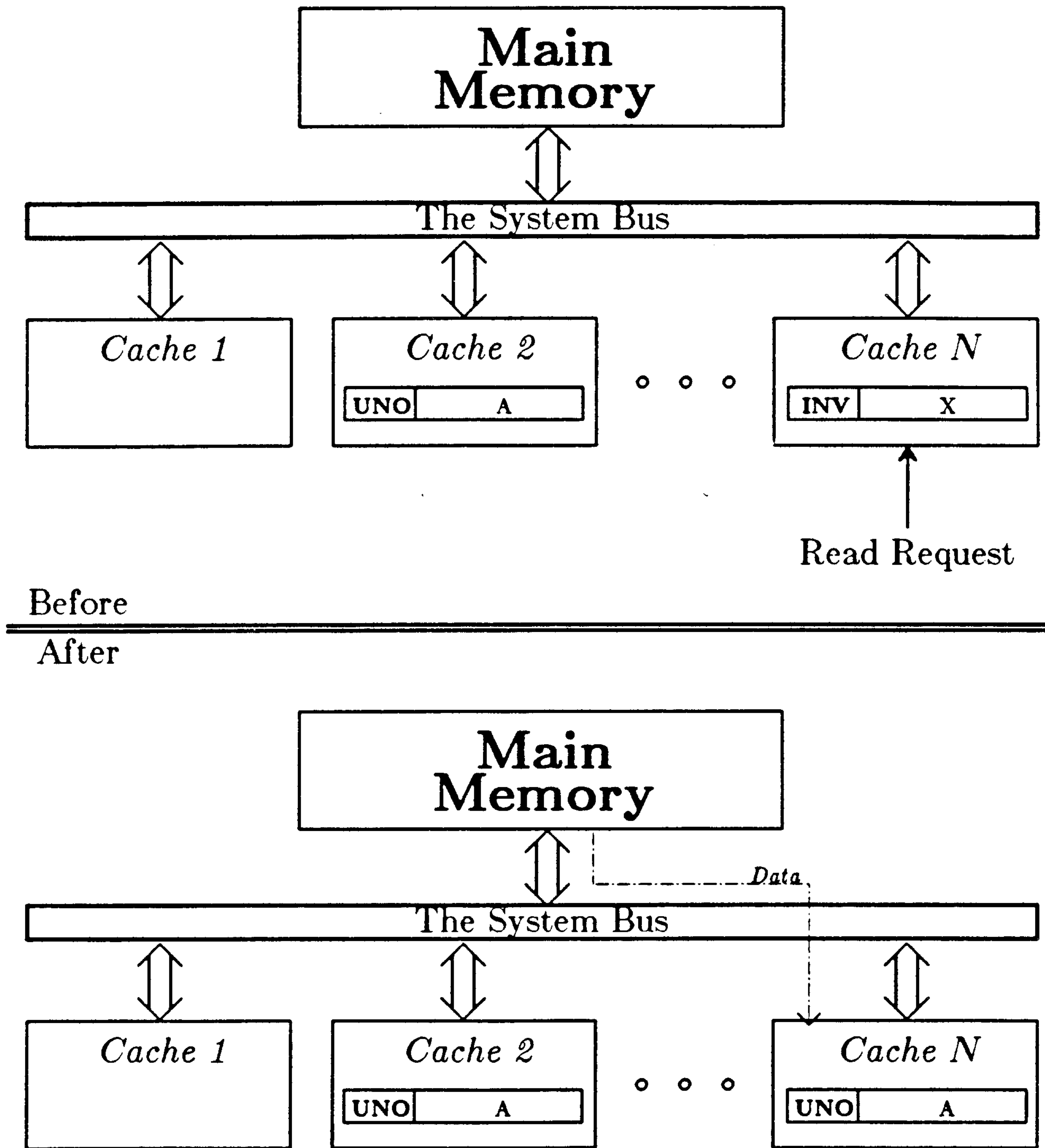


Figure 2.1 – Processor Read on Invalid Data

The Cache Controller in *Cache N* misses and sends out a *Read* request over the system bus to get the data. Main memory, the implicit owner here, replies to the request. The data ("A") travels along the system bus (as indicated by the broken line), and is placed into an entry in *Cache N*. This entry is then marked **UnOwned.** Note that although *Cache 2* had a copy of the data requested, it was not the owner and did not need to respond.

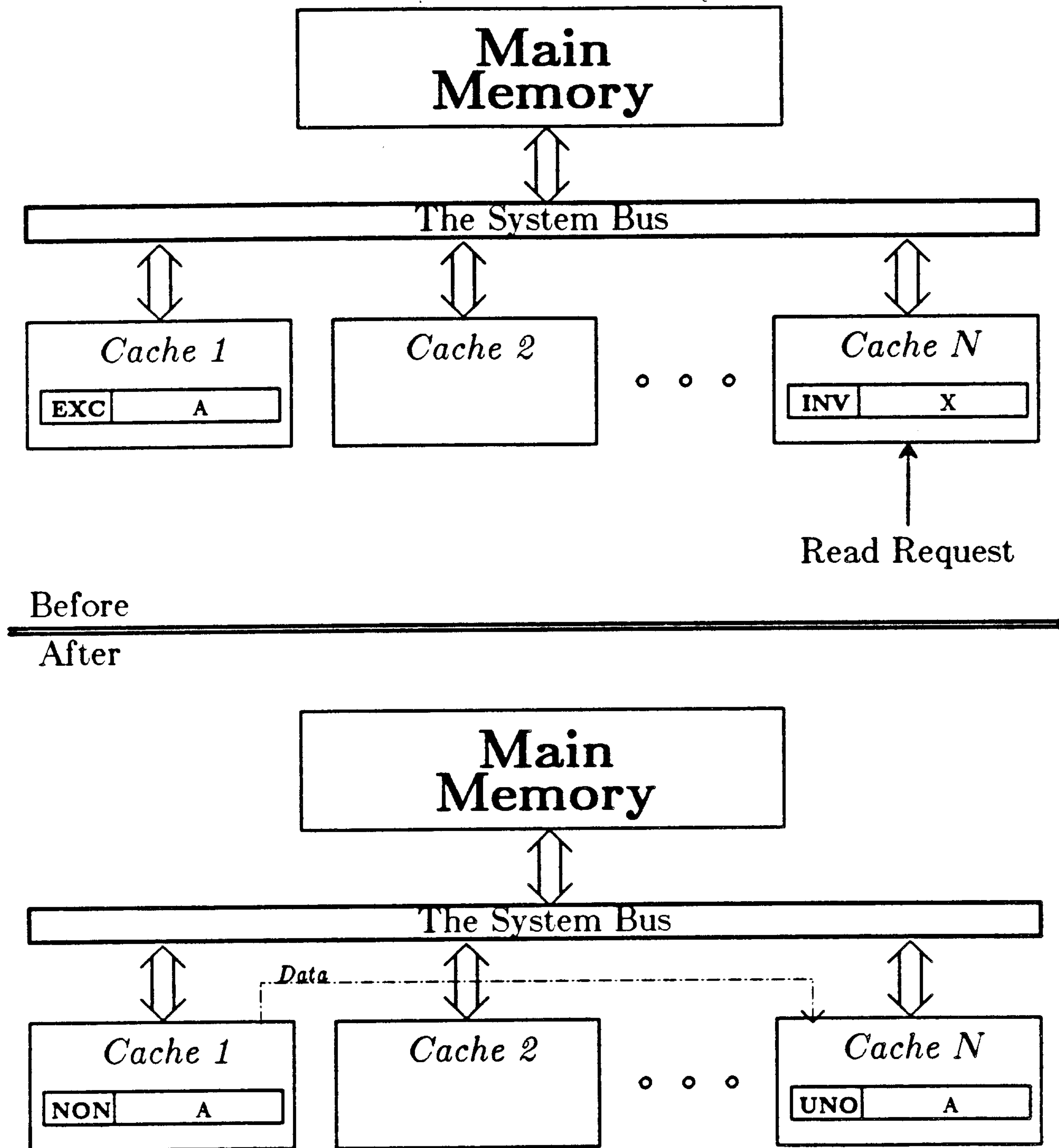
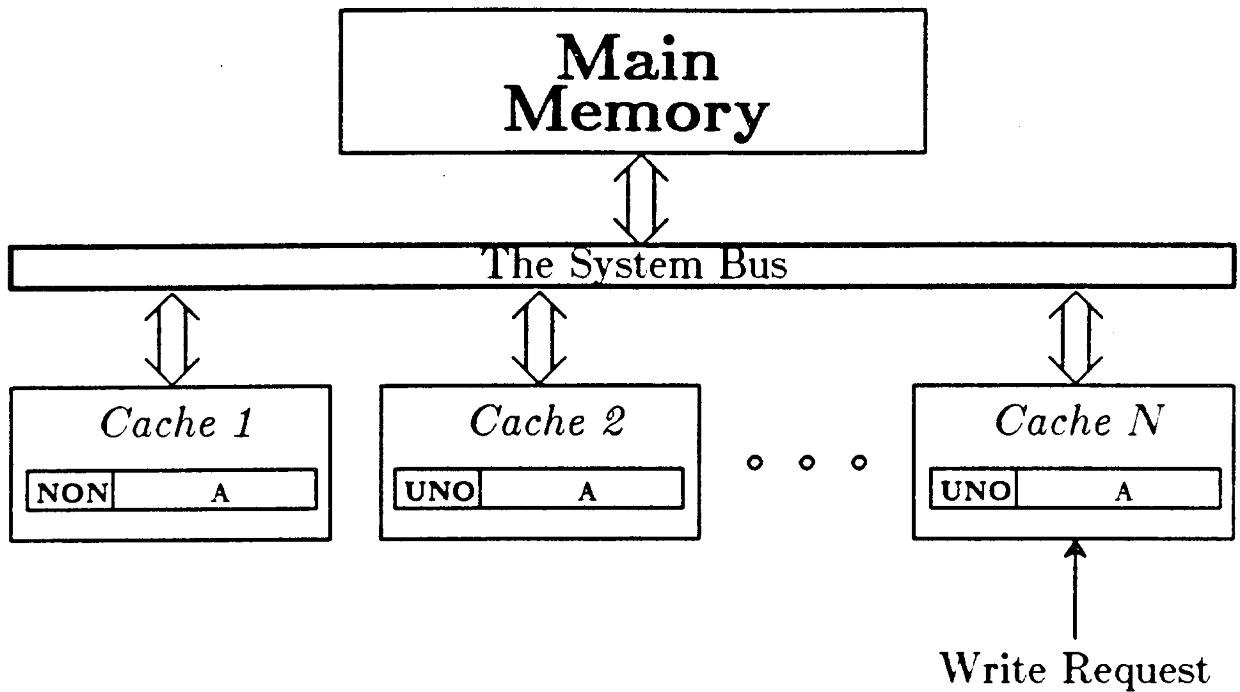


Figure 2.2 – Processor Read on Invalid/Exclusively-Owned Data

The Cache Controller in *Cache N* misses and sends out a *Read* request over the system bus to get the data. *Cache 1,* which owns the data, must respond to the request and supply the data. The data ("A") travels along the system bus (as indicated by the broken line), and is placed into an entry in *Cache N.* This entry is then marked UnOwned. Note that *Cache 1* must change its entry for "A" to *Owned NonExclusively,* since it no longer contains a unique copy of A.



After

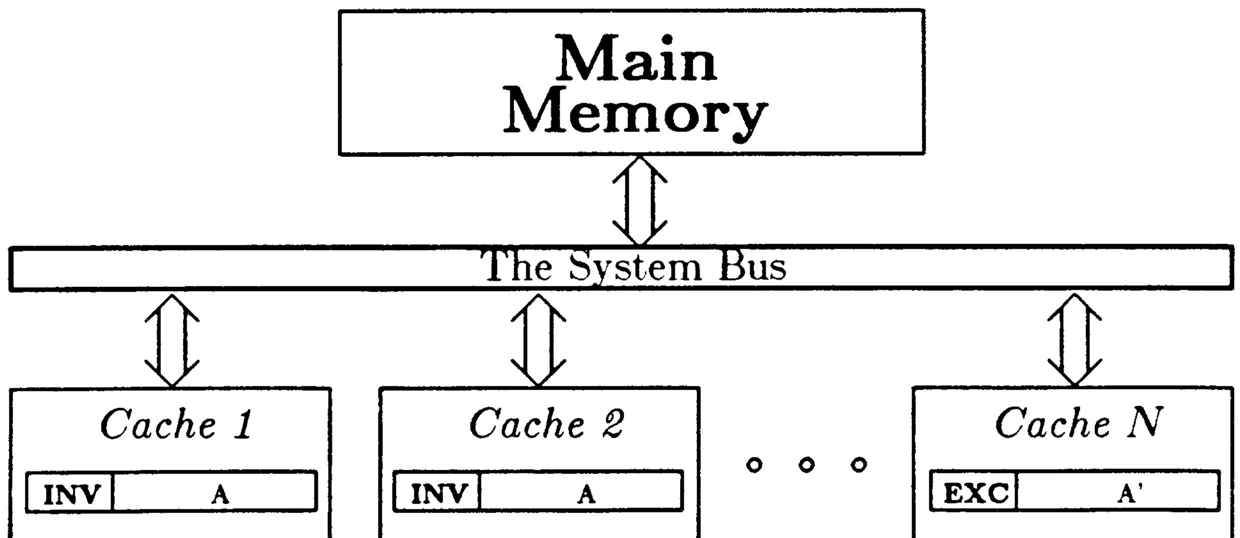


Figure 2.3 – Processor Write on Valid UnOwned Data

The Cache Controller in *Cache N* has a current copy of the data ("A"), and so does not need to perform a read. It "steals" ownership with a *Write-For-Invalidation*, which causes both *Cache 1* and *Cache 2* to invalidate their local entries containing A. No data travels along the system bus. *Cache N* modifies its entry locally (to A'), and the entry is then marked **Owned Exclusively**, since it contains a unique copy of A'.

cached copy of the block. The same procedure would be followed if the entry's state had been

Owned NonExclusively rather than **UnOwned.**

The last example, Figure 2.4, is again a *Write,* but Cache N does not have a copy of the block. Its Cache Controller must generate a *Read-For-Ownership* to obtain the data before it can update it. It then changes the entry's state to **Owned Exclusively** while the other caches invalidate their copies. Additional writes to this block can now be performed locally without any system bus traffic.

2.3. The Berkeley Ownership Protocol: Detailed Description

The protocol is implemented by two co-operating independent finite state machines: the Cache Controller and the Snoop Controller. The *Cache Controller* is primarily responsible for its own processor's use of the cache, interposing itself between the processor and memory. In addition, it assists in maintaining multiprocessor cache consistency: it must update the state of the cached block whenever it obtains or relinquishes ownership, and it must synchronize its actions with the Snoop Controller whenever it enters a critical section, e.g., to change an entry's state bits. The *Snoop Controller* is responsible for responding to the requests of other processors.

The critical assumptions we make about the system bus are: (1) it allows memory to be inhibited from servicing bus requests, (2) it can be extended with an additional line for protocol specific operations, and (3) at most one bus request is pending at a time. While these assumptions are true for the Intel Multibus, TI NuBus, Motorola VMEbus, and DEC Unibus, the latter assumption is not true for packet switched busses, such as the DEC SBIbus, the ELXSI Bus, and the Synapse Bus.

2.3.1. The Cache Controller

The Cache Controller's behavior depends on its processor's request, whether the data is in the cache, and the state of the cache entry on a hit. When a *processor read* results in a cache hit (see Figure 2.5), the appropriate word is provided to the processor. On a miss, the controller must first find an available cache entry, flushing data back to memory with a *Write-Without-*

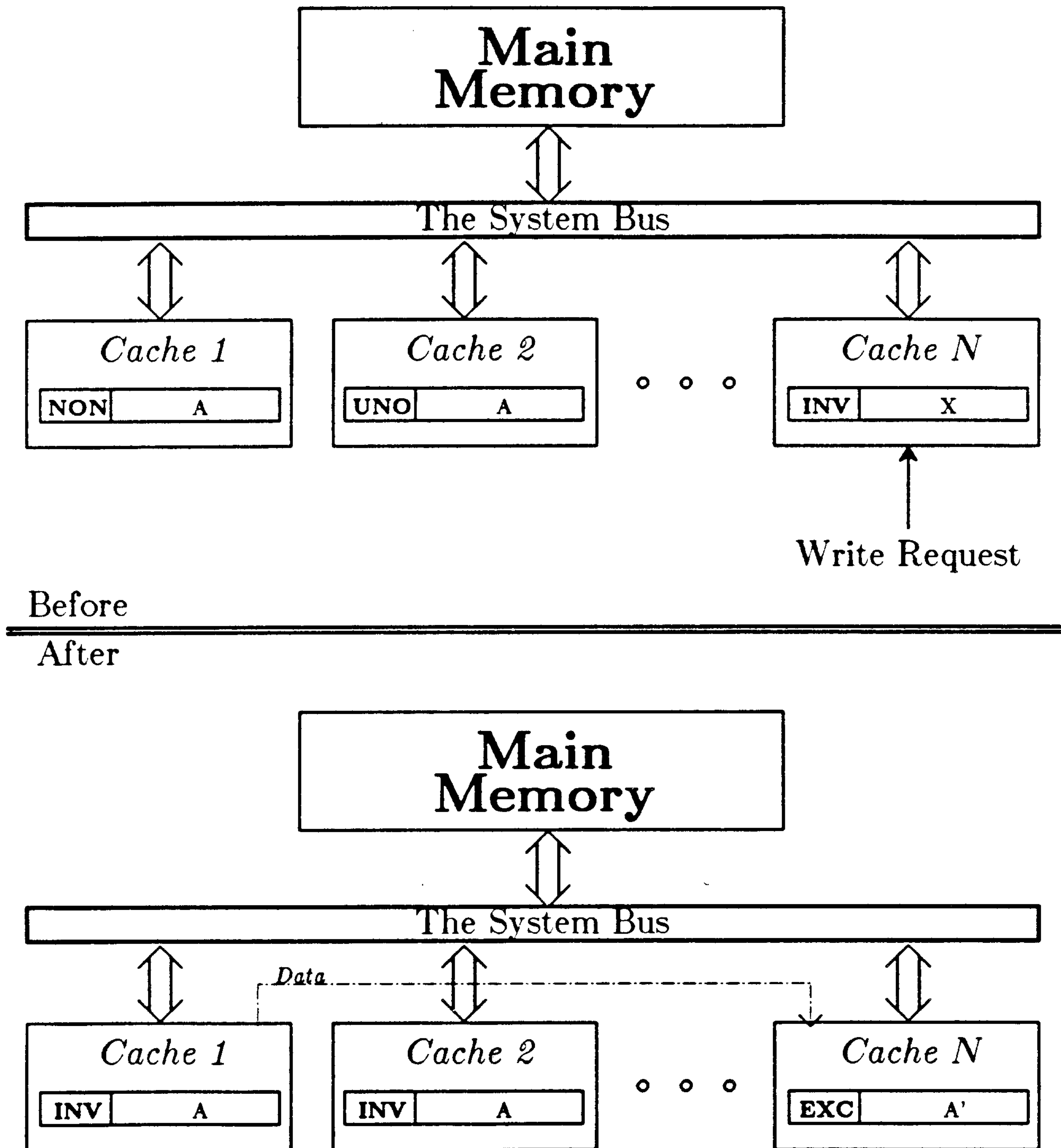


Figure 2.4 – Processor Write on Invalid Data

Cache N misses and needs to read the data ("A"). It must also modify this data, so it sends out a *Read-For-Ownership* request over the bus. *Cache 1*, the owner, responds (as shown by the broken line above), and sets its entry containing A to **Invalid,** because A is about to be updated elsewhere. *Cache 2*, noticing the ownership transfer, invalidates its entry for A. *Cache N* now updates the data locally (from A to A') and can then mark its entry **Owned Exclusively,** since it now contains a unique copy of A'.

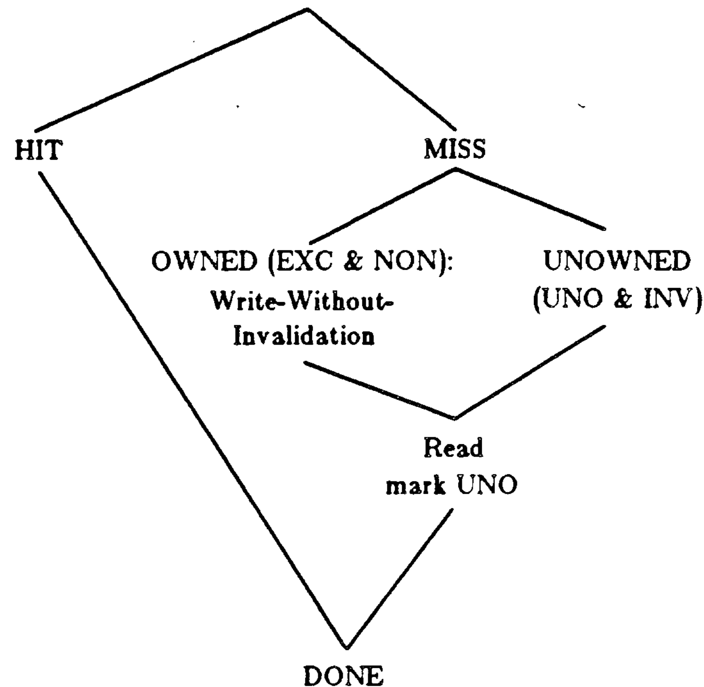


Figure 2.5 – The Cache Controller Handling a Processor Read Request
On a miss, a cache entry must be made available for the requested block. If the entry to be replaced is owned, then it must first be written (without invalidation) back to memory.

Invalidation if the replaced entry is owned. It then issues a *Read* for the desired block and changes its state to **UnOwned.** The actions across multiple cache controllers are serialized by acquiring exclusive access to the system bus. These interlocks are discussed in more detail in Section 3.4.

The procedure for a processor write to a block in the cache is shown in Figure 2.6. If the hit entry is **Owned Exclusively,** then the processor writes to it without broadcasting on the bus. However, the Cache Controller must first obtain exclusive control of the cache from the Snoop (see Section 3.4 for interlock details). If the state of the hit block is **Owned NonExclusively** or **UnOwned,** then the Cache Controller must signal to the Snoops of other processors its intention to write before it modifies the block, so the other caches can invalidate their matching entries. An

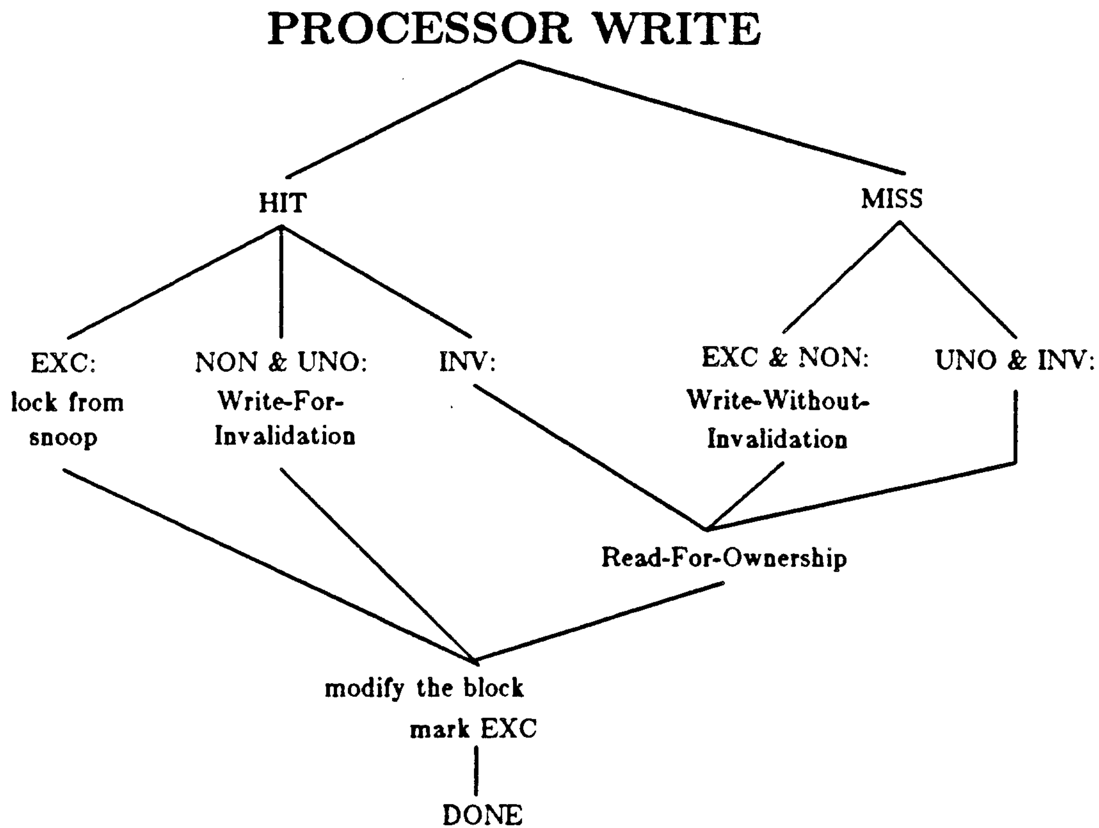


Figure 2.6 – The Cache Controller Handling a Processor Write Request

To service a processor write, the cache must first obtain ownership of the block. On a miss, it must acquire the block through *Read-For-Ownership*, possibly writing a replaced entry back to memory first. On a hit, ownership can be stolen for **Owned NonExclusively** or **UnOwned** entries by issuing a *Write-For-Invalidation*. **Owned Exclusively** entries can be updated without accessing the system bus.

Invalid state indicates that the Snoop invalidated the block in response to detecting a *Read-For-Ownership* or a *Write-For-Invalidation* from another processor, after the Cache Controller had initially detected a hit. This case is treated like a miss. On a miss, a block must be chosen for replacement. If the chosenosen block is owned, then it is written to memory using *Write-Without-Invalidation*. The requested block is then read with a *Read-For-Ownership* and is updated. The final state of the entry becomes **Owned Exclusively.**

2.3.2. The Snoop Controller

The *Snoop Controller* monitors the bus for reads and writes from other processors. It accesses its cache memory only to invalidate a block written by another processor or to provide an owned block for an external request. In the latter case, it also changes the entry's state from **Owned Exclusively** to **Owned NonExclusively.** The Snoop's actions are a function of the system bus request, whether it hits or misses in its cache, and the state of the entry.

If the Snoop detects a *Read* (see Figure 2.7), it first determines whether the block is in its own cache. If not, no Snoop action is necessary. If the address tag comparison hits, the Snoop's

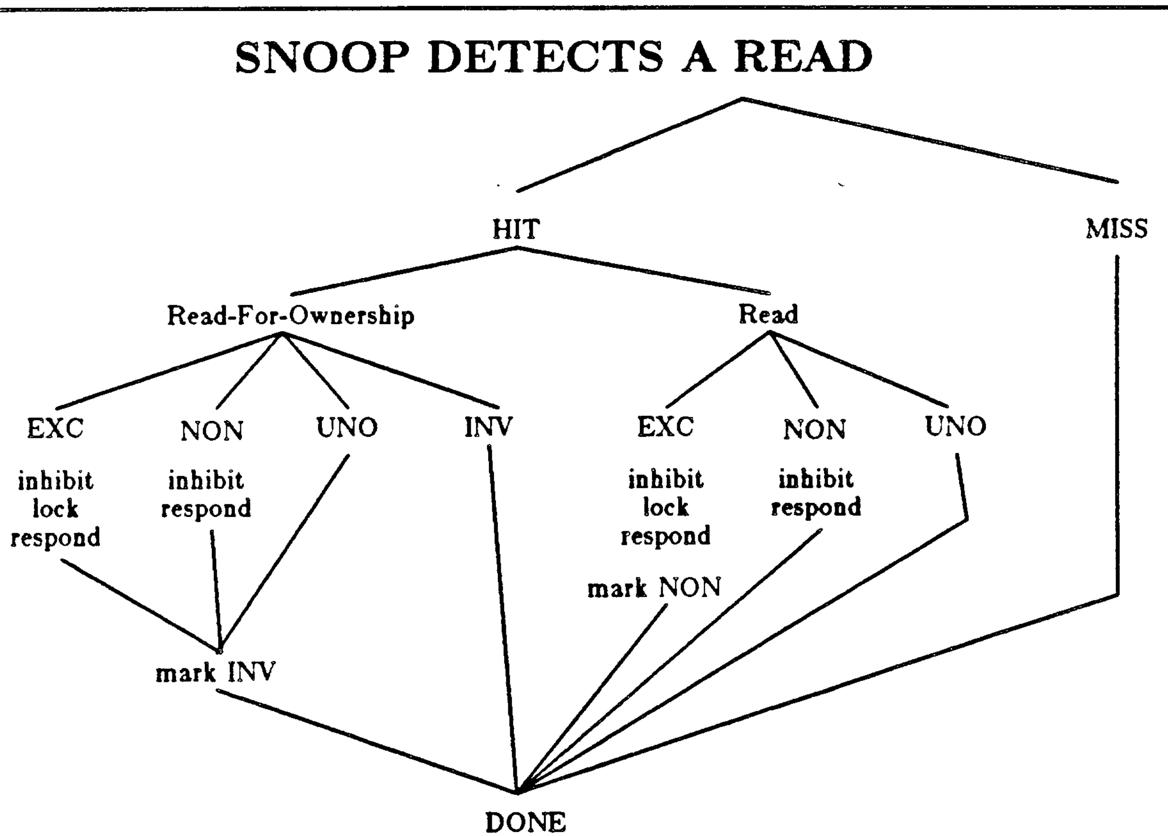


Figure 2.7 – The Snoop Controller Handling a System Bus Read Request

If the bus operation is a *Read-For-Ownership* that hits, the Snoop will deliver the block if it is **Owned Exclusively** or **Owned NonExclusively,** and will always invalidate the state. It behaves similarly for a *Read,* except that it does not invalidate the entry. An **Owned Exclusively** block that is read is changed to **Owned NonExclusively.**

response depends on the type of read *(Read* or *Read-For-Ownership)* and the state of the block hit (**Owned Exclusively, Owned NonExclusively,** or **Unowned**).[5] If the block is owned, the Snoop must inhibit memory from responding to the bus read and instead provide the data to the requesting processor. For a block that is **Owned Exclusively,** the Snoop must first obtain sole use of the cache memory before responding ("lock"), since the Cache Controller may attempt to simultaneously update the entry (see Section 3.4). If a *Read-For-Ownership* hits, then the Snoop must invalidate its copy. If a *Read* hits, the Snoop changes the block's state to **Owned NonExclusively** if it was previously **Owned Exclusively.** For the other state and bus operation combinations, no action is necessary; the owning Snoop will respond to the bus request.

The Snoop actions for a bus write are simpler (see figure 2.8). On a cache hit, the Snoop only responds in the case of a *Write* or a *Write-For-Invalidation*. This means that another processor or I/O device is updating a copy of the block, and the Snoop must invalidate its copy. If the write is a *Write-Without-Invalidation,* then another processor is flushing its cache, and the

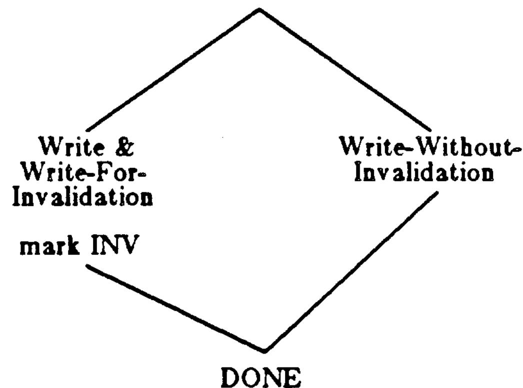


Figure 2.8 – The Snoop Controller Handling a System Bus Write Request
On a *Write* or *Write-For-Invalidation* that hits, the entry's state is marked **Invalid.** Otherwise, the Snoop does nothing.

[5] A hit on a block marked **Invalid** is treated as a miss.

Snoop need do nothing.

2.4. Comparison and Evaluation

This section compares the performance of the Berkeley Protocol with write-first [GOOD83, RAVI83] (write-through [AGRA77] is quickly dismissed). We distinguish between the ownership protocol as described so far (the *naive* protocol) and a more sophisticated version that takes advantage of processor hints in order to improve performance. The essential differences are (1) processor instructions that distinguish between loads of shared and non-shared data, and (2) the incorporation of clean/dirty block information in the state.[4]

Both ownership and write-first employ *copy-back*. Dirty blocks are retained in the cache as long as possible; main memory is only updated when the block is flushed from the cache. Write-through places an upper bound on the total number of processor bus writes, since such a strategy generates a bus write for each cache write. Copy-back strategies generate fewer bus writes, thus reducing the demand for the limited bus bandwidth.

Consider the costs for the ownership and write-first protocols for the following operations: reads, single writes, and multiple writes to the same memory block, which can be either shared or non-shared. We observe that in a typical job mix, reads will dominate writes and that most sharing will be read-only, with little write sharing [MONT77].

A weakness of the naive ownership protocol is that it does not distinguish between processor accesses to shared and non-shared data. The cache controller cannot predict whether an acquired block will eventually be updated. Thus, it must acquire block ownership at the time of the processor write. As shown in Table 2.3, a single write to non-shared data represents the worst case for the naive protocol. However, if the processor has an instruction to load non-shared data, then the more sophisticated protocol can always acquire such data with a *Read-With-Ownership* request. Thus, the additional operation to obtain ownership at the time of the write can be

[4] In the naive protocol, a block is acquired with ownership only when it is to be updated. Thus there is no need for a separate dirty bit. Since this is not the case for the sophisticated protocol, a dirty bit is needed to avoid unnecessary write-backs.

Table 2.3 — Comparison of Ownership with Write-First (Non-Shared)			
Operation	**Ownership (sophisticated)**	**Ownership (naive)**	**Write-First**
Non-Shared Read	Read-For-Ownership	Read (UnOwned)	Read
Non-Shared Single Write	Read-For-Ownership Write-Without-Invalidation	Read (UnOwned) Write-With-Invalidation Write-Without-Invalidation	Read Write-Thru
Non-Shared Multiple Writes	Read-For-Ownership Write-Without-Invalidation	Read (UnOwned) Write-With-Invalidation Write-Without-Invalidation	Read Write-Thru Write-Back

avoided.

In the write-first protocol, the first write is through to memory to invalidate copies of the block in other caches. This write is incurred *even though the data is not shared!* However, in the case of a single write to a block, this write is not additional, since the data must eventually be flushed from the cache. It represents a wasted write only when there is more than one processor write to the same block, since the additional final write to memory is still needed.

An analysis of several representative traces[5] indicates that a large number of cache blocks do receive multiple writes, and that the number of bus operations per processor is significantly larger for write-first than for ownership. For example, a 64K cache with a 64 byte[6] blocksize will generate from 11 – 22% more system bus operations (depending on the trace) if it implements write-first instead of ownership. For a more conventionally sized cache, e.g., the VAX 780's 8K cache with 8 byte blocks, the write-first protocol generates from 20 – 27% more bus operations. These additional operations are incurred *per processor* in a multiprocessor system, *even when no data is shared!* The effect on the limited bus resource is as though an additional (useless) processor were added to the bus for each of five processors that contribute to system throughput.

[5]The benchmarks are (1) the compilation of the Puzzle program executing on a RISC simulator, (2) the same compilation executing on the VAX, (3) the execution of Vaxima on the BEGIN benchmark, and (4) a self-compilation of Franz Lisp.

[6][KATZ84] justifies why this is a good cache configuration for a multiprocessor.

It is also advantageous if the processor can distinguish between access to read-only and writeable data to shared data. Then the cache could access read-only data with a *Read* request, rather than a *Read-For-Ownership*. Using the latter defeats the purpose of read sharing, since acquiring ownership simultaneously invalidates copies of a block in other caches. Otherwise, the behavior of the protocols is identical to the non-shared case.

A comparison of the dynamic behavior of the protocols is more difficult to obtain, because of its dependence on actual execution histories. Consider the case of interleaved reads and writes to a block that is continually read and updated, as might well be the case for a test-and-set lock. In the ownership protocol, the block is acquired with *Read-For-Ownership* and is modified in the cache. Suppose that it is not accessed again before it is replaced from the cache and is flushed to memory. In that case, the number of bus operations is the same as if there is no sharing (i.e., a *Read-For-Ownership* followed by *Write-Without-Invalidation*). However, requests for this block occur frequently, and it is likely that it will pass to another cache before being replaced to memory. Thus, from the viewpoint of a single cache, updating the block requires only one bus action: the initial read for ownership. On the other hand, in the write-first protocol, the block is read, modified in the cache, and then written to memory to invalidate other copies. In this case, write-first requires two bus actions to the ownership protocol's single action. In general, the ownership protocol tends to transfer blocks directly between the caches, whereas write-first transfers blocks from the original cache to memory to the other caches, incurring the full latency of the memory system. (Note that the motivation for the write-broadcast protocol of [RUDO84] is to remove some of these deficiencies from write-first).

In summary, since a substantial number of blocks are written more than once, *the sophisticated version of ownership clearly dominates write-first*. Further, it should not be difficult for a compiler to obtain the necessary hints about non-shared, shared read-only, and shared writeable data from declarations in the program.

3. Chip Implementation

3.1. Chip Architecture

The snooping data cache system consists of five distinct subsystems (see Figure 3.1): a *Data Cache Memory*, a (processor) *Cache Controller*, a *Snoop Controller*, a *Processor Bus Interface*, and a *System Bus Interface*. The functions of the Cache Controller and Snoop Controller have already been described. The *Cache Memory* is organized as a direct mapped array of 64-bit cache blocks with associated state and tag bits. This organization was chosen for its simplicity. Additional circuitry includes tag match logic to determine cache hits, and assembly registers to

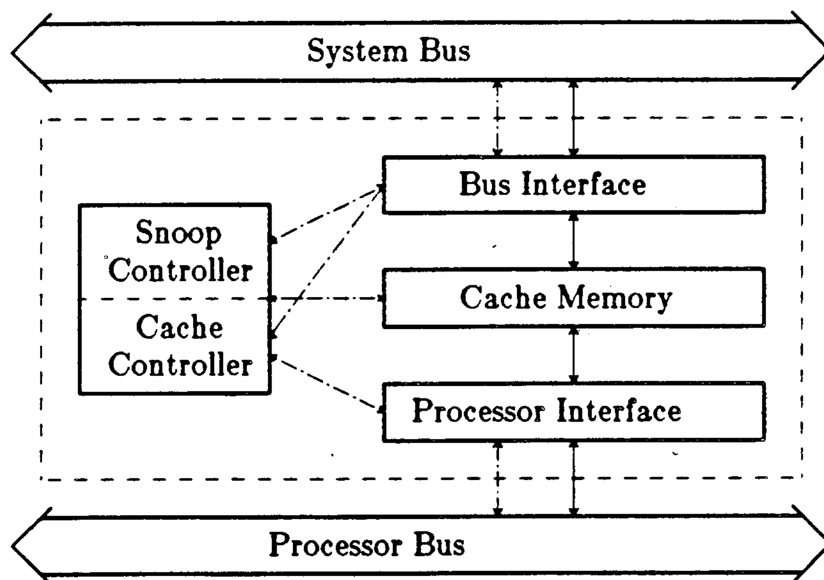


Figure 3.1 – A Block Diagram of the Data Cache Chip

The data cache chip is partitioned into the subsystems illustrated above. Data lines are solid arcs; control lines are broken. Data transfer between the cache memory and the processor is performed by the Processor Bus Interface subsystem. Data transfer between the cache memory and the system bus is performed by the System Bus Interface subsystem. The Cache Controller handles the actual processor reads and writes on the cache. The Snoop Controller watches the system bus and implements much of the cache consistency protocol.

map 16-bit external data[7] to and from 64-bit internal data blocks. The *Processor Bus Interface* implements the handshake with the processor. The *System Bus Interface* encapsulates the details of communicating with the system bus. While only the Cache Controller needs access to the Processor Bus Interface, both controllers must interface with the system bus.

3.2. The Cache Memory Subsystem

The chip's datapath is the cache memory subsystem (see Figure 3.2). The cache itself is organized into sixteen entries, each entry containing a sixty-four bit data block (4 x 16-bit words), two state bits, and thirteen tag bits. The state bits can be modified independently of the data and tag portions of the cache. The datapath contains two sets of decode circuitry (Adec and Bdec). On a read, these can be driven independently (dual-port read); on a write they are driven by the same inputs (single-port read).

Data is read from and written to the cache from two 64-bit assembly registers (Aassembly and Bassembly). On a read, these registers deliver a 16-bit portion of the data to the system bus or an 8- or 16-bit portion to the processor bus. On a write, they assemble four 16-bit words from either the processor or the system bus into one 64-bit block and then transfer it to the cache. The Processor Bus Interface is directly connected to the Bassembly register. The System Bus Interface is directly connected to the Aassembly register.

The cache controller decodes nineteen bits of a word address provided by the processor. The low order two bits (Adr[2..1]) select a word from within the four word memory block. These address lines control sixteen 4:1 multiplexors/demultiplexors on each side of the data portion of the cache memory. The next highest four bits (Adr[6..3]) select one of the sixteen entries of the cache, and are input to the decoders. The highest order thirteen bits (Adr[19..7]) are compared with the tag portion of the selected entry. If they match, and if the state is not **Invalid**, then the addressed word is *hit* in the cache and the appropriate match line is asserted *(MatchA* or

[7]16-bit external data paths were chosen for compatibility with existing microcomputers and system busses, in particular, the Motorola 68010 and the Intel MultiBus.

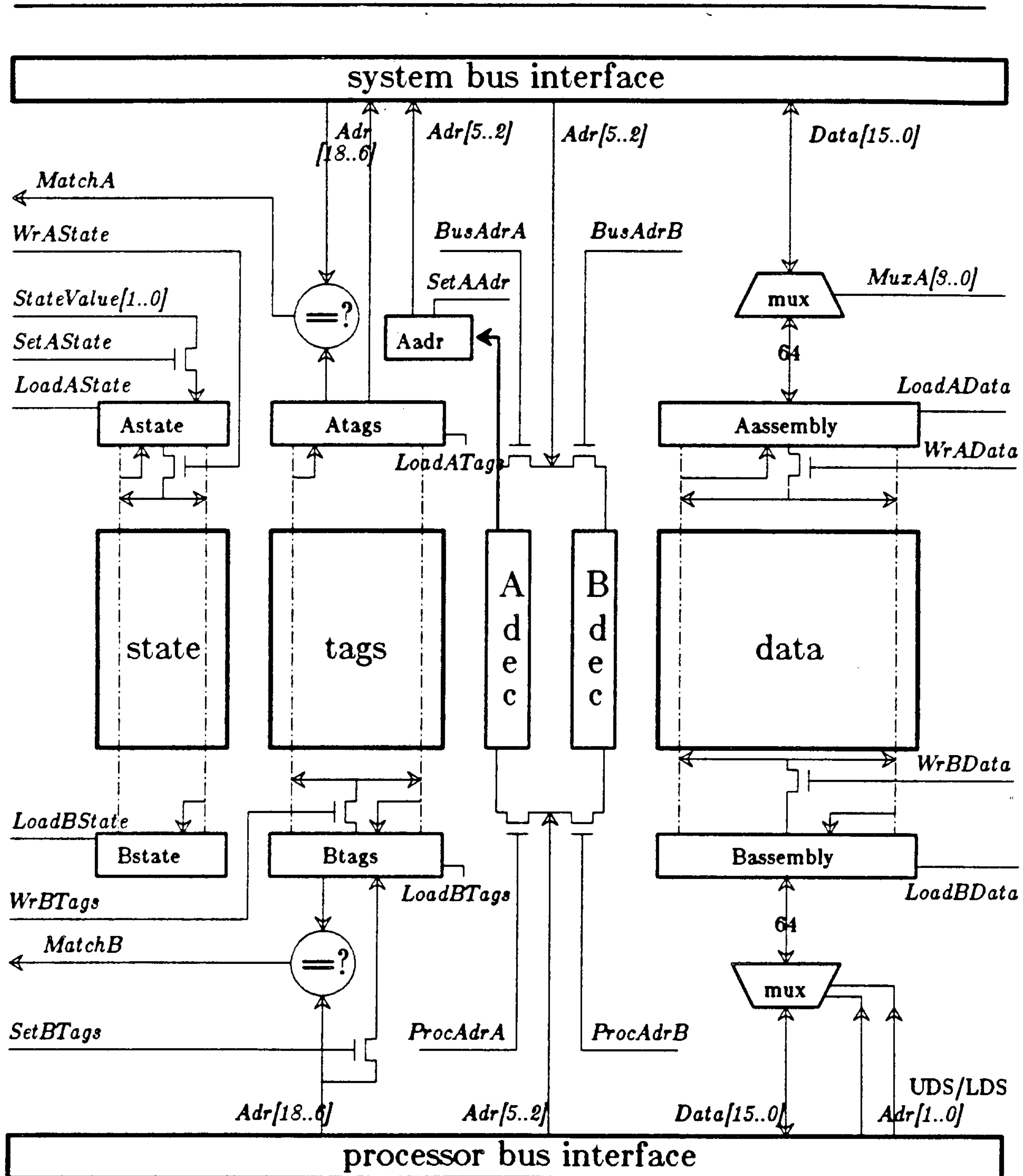


Figure 3.2 – The Datapath Block Diagram & Control Signals

This diagram shows the major units of the cache datapath and the signals used by the Snoop and Cache Controllers to manipulate it.

MatchB); otherwise it is a *miss*.

The datapath is accessed via two mutually exclusive read and write cycles. Each cycle employs a standard two-phase clock. The cycles are distinguished by a special *WriteCycle* signal, asserted to initiate a write cycle. The clock phases operate in an identical fashion for both cycles. During the first phase, *Phi1,* the decoders and word lines are driven. During the second phase, *Phi2,* the bit-lines are driven either by the cache cells or the assembly registers. The datapath timing influences the timing of the entire chip. The simple two phase clock, derived from the processor's clock, is used throughout. Every subsystem obeys the standard convention of inputs sampled on *Phi1,* outputs valid on *Phi2.*

3.3. The Cache and Snoop Controllers

The signals used by the controllers to access the memory subsystem are also shown in figure 3.2. To respond to a processor (system bus) read request, the Cache (Snoop) Controller initiates address decode by asserting *ProcAdrB (BusAdrA)* on *Phi1.* The Cache (Snoop) Controller reads into the B (A) registers by asserting the *LoadBData, LoadBTags,* and *LoadBState (LoadAData, LoadATags, LoadAState)* signals on *Phi2.* When writing the data and tag portions of the cache, the Cache Controller asserts both decode signals *(ProcAAdr* and *ProcBAdr),* and writes with *WrBData* and *WrBTags.* The Snoop never updates these portions of the cache; however, either controller can change the state. This requires an extra phase to set *(SetAState)* the new state value *(StateValue)* into Astate.

3.4. Interlocks Between the Controllers

Because the Snoop and Cache Controllers operate independently, they contend for access to the cache memory. Read-read, read-write, and write-write contention are possible. The datapath has been implemented with a dual-ported read capability to eliminate read-read contention. However, since the datapath is single-ported for write, only one controller can be active during write phases. Read-write contention is eliminated by requiring the controllers to adhere to the

separate read cycle/write cycle protocol for accessing the cache memory.

To illustrate how read-write contention could arise, consider the case where the processor issues a read request that hits in the cache, and simultaneously, the Snoop detects a *Write-For-Invalidation* for the same block. The Snoop must change its entry's state to **Invalid,** while the Cache Controller needs to read from the cache. Because reads and writes occur on different cache cycles, the Snoop invalidates the entry only after the Cache Controller has read it, or vice versa.

Write-write conflicts occur when both controllers must update the cache at the same time. For example, the Snoop may need to invalidate an entry while the Cache Controller steals ownership by changing an entry's state from **UnOwned** to **Owned Exclusively.** Besides these *intra-cache* write conflicts, there are other *inter-cache* update anomalies, as Figure 3.3 shows.

Two interlock schemes can be used to prevent write-write conflicts. The most straightforward tactic is to use the system bus as a semaphore. In this case, before the Cache Controller can update the cache, it must arbitrate for and acquire control of the system bus. Once in control, no other cache can generate requests on the bus, and therefore its Snoop is inactive. The bus is released only after the update is complete.

Using the system bus as a semaphore has severe performance implications. Once a Cache Controller has obtained the bus, other Cache Controllers are prevented from using it until it has been released. This is acceptable only if the first Cache Controller needs access to the bus anyway, to satisfy its processor's write (the block is not in the cache and must be read) or to signal a *Write-For-Invalidation*. This is not the case for writes to **Owned Exclusively** blocks, since these can be done locally and do not require the bus. (Figure 3.4 shows how a conflict can arise when a processor writes to an **Owned Exclusively** block, and there is a simultaneous external read request for the same block.)

An additional interlock mechanism is needed for writes to **Owned Exclusively** blocks (see Figure 3.5). The interlock is asymmetric to favor local writes to owned blocks over external reads. The Cache Controller wins ties because it can obtain the lock in the same cycle it is

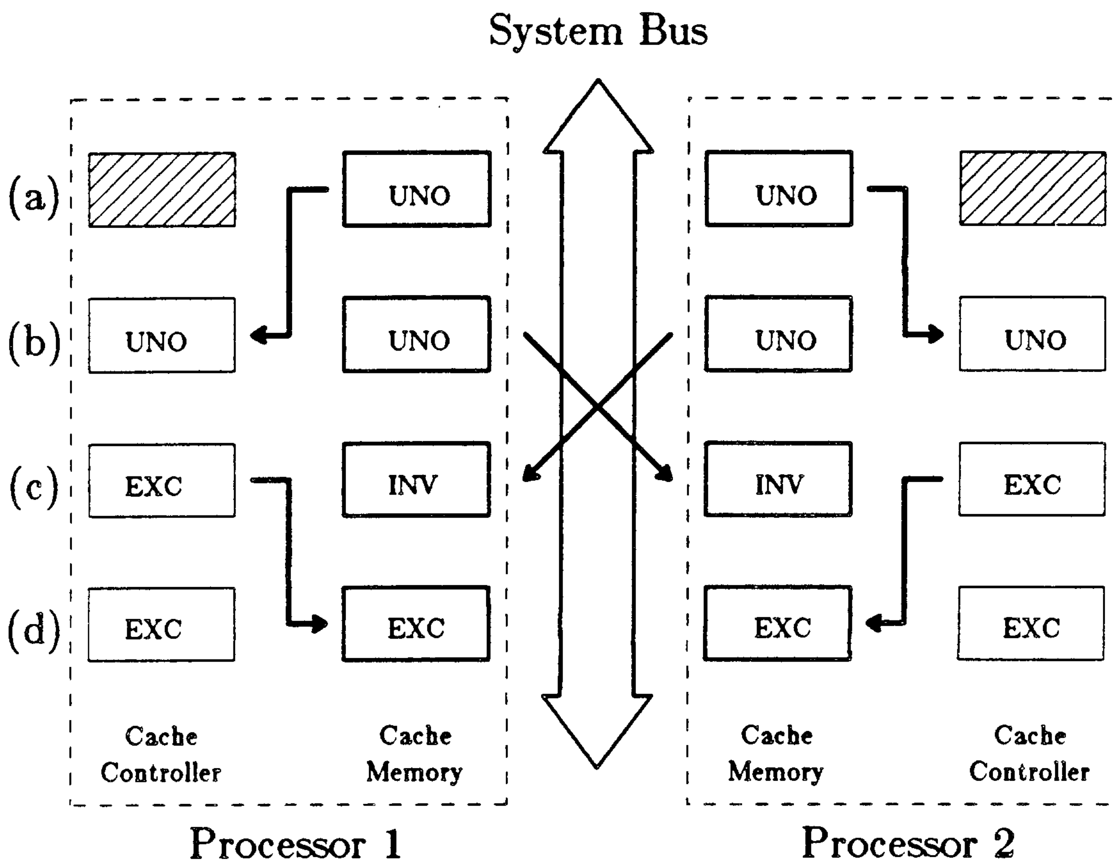


Figure 3.3 – Example of an Update Anomaly

In (a), processors 1 and 2 each have **UnOwned** copies of the same memory block to which they wish to write. Under the Berkeley protocol, they must obtain ownership by issuing a *Write-For-Invalidation* before the writes can be performed. In (b), the Cache Controllers simultaneously read and discover that the entries are **UnOwned.** In (c), they update the state to **Owned Exclusively**, and (in sequence) send *Write-For-Invalidation* requests to the system bus. The respective Snoops invalidate the entries in their caches. In (d), the controllers now update their caches, which they still believe to be **Owned Exclusively.** An inconsistent state has been reached.

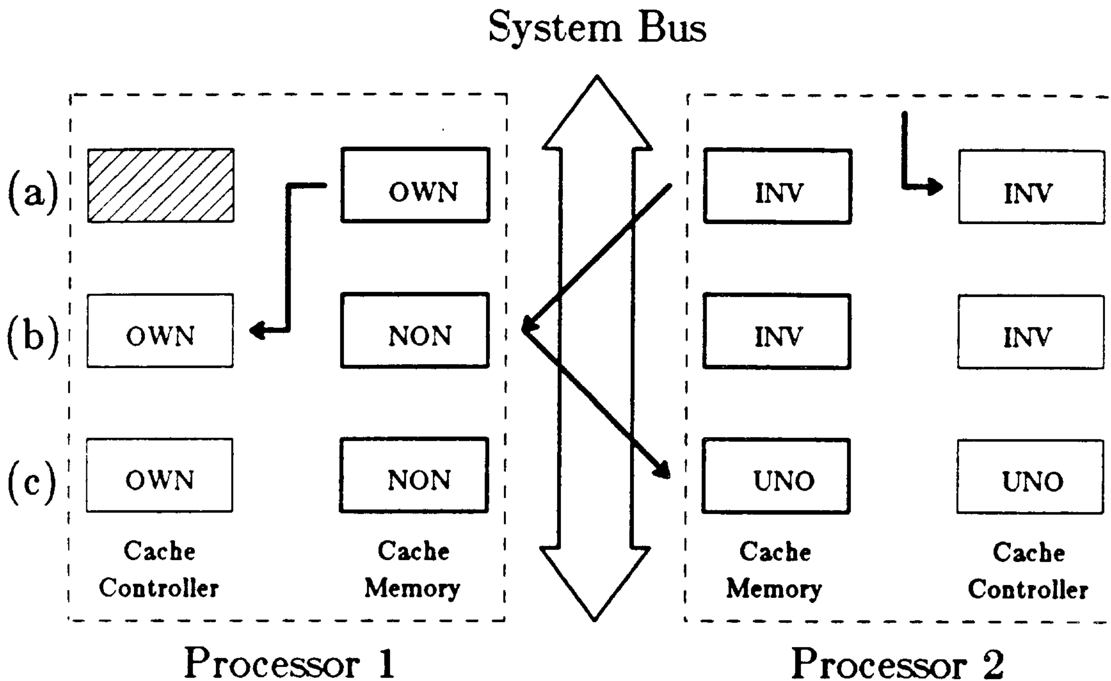


Figure 3.4 – Conflicts with Blocks that are Owned Exclusively

In (a), processor 2's Cache Controller tries to read a block in its cache and misses. Its Cache Controller then issues a *Read* for the block. In (b), processor 1 accesses the same block and discovers that it has a state of **Owned Exclusively**. It therefore believes it can update the entry. Simultaneously, its Snoop responds to processor 2's *Read* request, placing the block's data on the system bus and changing its entry to **Owned NonExclusively**. In (c), the cache controller for processor 2 changes the state of its copy to **UnOwned**. The Cache Controller of processor 1 then updates its copy, causing an inconsistent state.

needed, while the Snoop must request it a full cycle in advance. There is a one cycle latency whenever another cache accesses an **Owned Exclusively** block.

4. Analysis of the Cache Consistency Protocol Implementation

In this section we analyze the implementation of the cache consistency protocol to insure that all race conditions are properly behaved and that the implementation maintains cache consistency. The major item of contention is the cache memory. In general, race conditions between the Cache Controller and the Snoop are avoided when the Cache Controller has control of the system bus, thus disabling the Snoop.

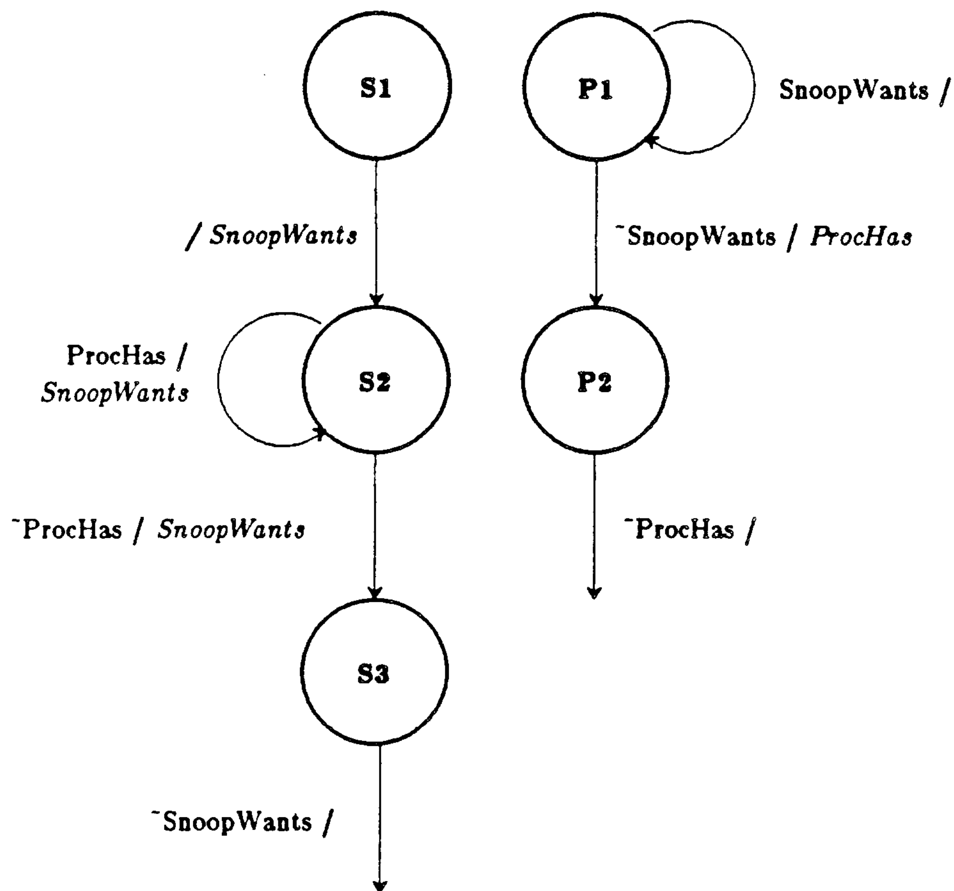


Figure 3.5 – Asymmetric Interlock Scheme

The state diagram above shows how the processor (P) and the snoop (S) should act to avoid entering their criticial sections at the same time. This interlock is only needed when an entry must be updated and it is **Owned Exclusively,** since otherwise the processor would have obtained the system bus, thus effectively locking out the Snoop. The ProcHas line can be asserted by the processor right away, as long as SnoopWants is not already asserted, and the processor is safe while in state P2. The SnoopWants line can be asserted any time, but the snoop must wait a cycle and check ProcHas before entering its safe state, S3. Note that the processor is purposely given the upper hand, since an ordering where the snoop wins first causes some wasted processor actions (such as rereading the data).

Consider the case of a *processor read*. If the block is not in the cache, it must be read from memory. The Cache Controller acquires the system bus, thus disabling the Snoop. The interesting case occurs when the block is in the cache, but the Snoop is attempting to invalidate it in response to an external operation. If the Cache Controller is reading, then the separate read/write cycle interlock insures that the Snoop cannot be writing, and vice versa. The Cache

Controller will either see an invalidated block or its value before it was invalidated, but nothing "in-between". Thus consistency is maintained.

Now consider the case of a *processor write* to an **Invalid, UnOwned,** or **Owned NonExclusively** block. In each of these cases, the system bus must be acquired before the *processor write* is actually serviced. After determining the state of the block and acquiring the system bus, the Cache Controller must reread the state to insure that the Snoop has not invalidated the block while it has been arbitrating for the bus. The bus must be held until the cache has been updated to avoid livelock. Otherwise, a cache can acquire the block, only to have it stolen by another cache. The cache will immediately try to steal the block back, only to have the process iterate indefinately.

The final case is a *processor write* to an **Owned Exclusively** block. We need not worry about an external *Write-For-Invalidation* affecting this block, since only a cache with a valid copy may issue a *Write-For-Invalidation,* and there are no other copies. The interesting situation occurs when the external operation is a *Read* (with or without ownership). Since the implementation for these requests are similar, their treatment can be combined.

The critical section for the Snoop begins with reading the block from the cache memory and continues through to storing the new state. The critical section for the Cache Controller begins with reading the state (to insure that the block is still **Owned Exclusively**) until writing the modified block back into the cache. The race condition is that the Snoop supplies the unupdated block to the requestor, updating the state to either **Invalid** or **Owned NonExclusively,** while the modified block is written into the cache.

The detailed Snoop behavior is:

* Read the block (state [Owned Exclusively], tag, and data)
* Raise the bus INHIBIT signal
* Obtain intra-cache interlock
==START CRITICAL SECTION==
* Re-read the data
* Place the appropriate word on the bus and raise ACK
* Store the new state (Owned NonExclusively or Invalid)

* Release intra-cache interlock
==END CRITICAL SECTION==
* Lower INHIBIT

The detailed behavior of the Cache Controller is:

* Read the block (state [Owned Exclusively], tag, and data)
* Obtain the intra-cache interlock
==START CRITICAL SECTION==
* Re-read the state
* If the state is still Owned Exclusively, then
  - Form the modified block in the assembly register
  - Write the block into the cache memory
  - Release the intra-cache interlock
* If the state is not Owned Exclusively, then
  - Release the intra-cache interlock
==END CRITICAL SECTION==
  - Restart the write-request as for an Invalid or Owned NonExclusively block

As a final note, we have found it straightforward to implement an atomic *Test & Set* operation within the cache. It is handled like a *processor write* to an **Owned Exclusively** block. The block is read from the cache, the selected word is set to a "1", and the original word is delivered to the processor. The interlocks ensure that the operation is executed atomically.

5. Summary and Conclusions

We have presented a snooping cache consistency protocol that improves on the write-first protocol in the common case of multiple writes to non-shared data. This improvement is achieved by requiring the processor to give information about the kind of data it is acquiring for its processor: is it non-shared, read-only, or will it be updated in the future? This suggests that new instructions, e.g., "load non-shared," should be considered for processors that are used as multiprocessor building blocks. The protocol also leads to an efficient implementation of a processor *Test & Set* operation within the cache.

The described consistency protocol has been realized as a single chip system implemented in 3 micron P-Well CMOS technology. The implementation effort allowed us to identify both the inter- and intra-cache race conditions, and simulations of the chip level implementation have been used to verify the correctness of the protocol. The single chip system has been designed for the

Motorola 68010 microprocessor with the Intel MultiBus as the system bus (extended with an extra bit to distinguish the write type). Unfortunately, the chip did not meet its required timing specifications, imposed by the 10 Mhz clock rate of the 68010, and its control portion is currently under redesign. The complete implementation details can be found in [BORR84]. A plot of the partially completed chip is given in Figure 5.1.

Our work provides some insights into the implementation complexity of write-broadcast protocols. The chip's datapath already supports writing the cache memory from either the system bus or processor bus sides. The system bus assembly register can be used to assemble the block and to write it into the cache under Snoop control. The added complexity of the write-broadcast protocol is the design of a broadcasting Cache Controller. To maintain cache consistency, it must hold the bus while broadcasting until all caches have completed updating their caches. The difficulties of implementing reliable broadcast protocol are well known, and are not directly supported by any system bus known to us.

We wish to acknowledge the efforts of Gaetano Borriello (system bus interface), Harry McKinley (processor bus interface), Walter Scott (interlock design), and Shaun Whalen (datapath design) for the original design of the snooping cache protocol and its prototype implementation, and Ho-Ming Leung for the controller redesign. Mark Hill, David Patterson, and George Taylor read earlier drafts and provided excellent comments.

6. References

[AGRA77] Agrawal, O. P., A. V. Pohm, "Cache Memory Systems for Multiprocessor Architectures," Proc. AFIPS National Computer Conference, Dallas, TX, (June 1977).

[ARCH84] Archibald, J., J-L Baer, "An Economical Solution to the Cache Coherence Problem," Proc. 11th Ann. Symp. on Computer Architecture, Ann Arbor, MI, (June 1984).

[BORR84] Borriello, G., S. Eggers, R. Katz, H. McKinley, C. Perkins, W. Scott, R. Sheldon, S. Whalen, D. Wood, "Design and Implementation of an Integrated Snooping Data Cache," UCB/CSD Technical Report, (August 1984).

[KATZ84] Katz, R. H., D. A. Patterson, S. J. Eggers, G. A. Gibson, P. M. Hansen, M. D. Hill, J. M. Pendleton, S. A. Ritchie, G. S. Taylor, D. A. Wood, "Multiprocessor RISCs: Design Issues and Initial Analyses," submitted to 12th Annual Symposium on Computer

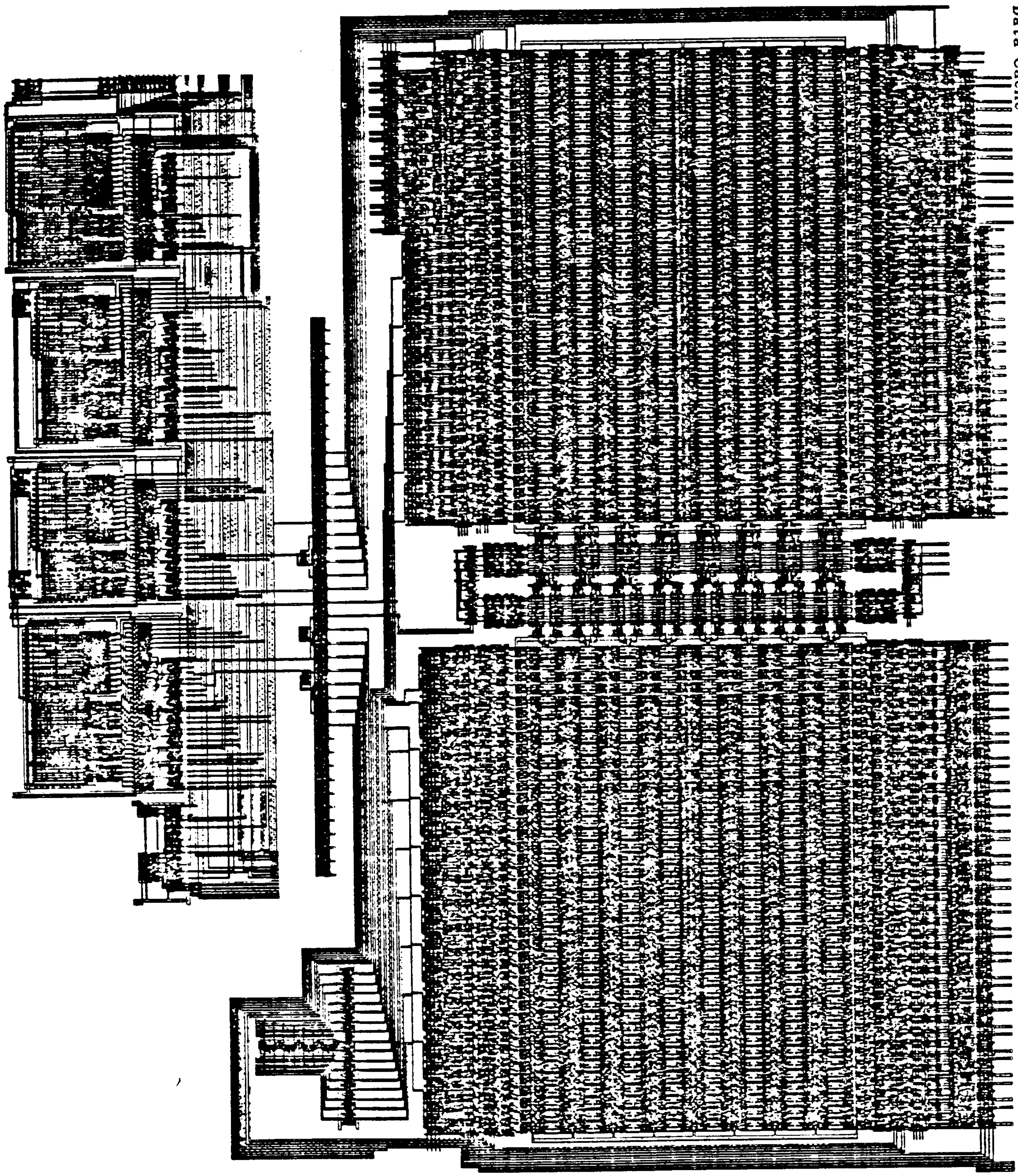


Figure 5.1 – Checkplot of Partially Completed Snooping Cache Controller

Architecture, Boston, MA, (June 1985).

[FRAN84] Frank, S., "Tightly Coupled Multiprocessor System Speeds Memory-Access Times," *Electronics,* (Jan 12, 1984).

[GOOD83] Goodman, J., "Using Cache Memories to Reduce Processor-Memory Traffic," 10th Annual Symposium on Computer Architecture, Trondheim, Norway, (June 1983).

[MCCR84] McCreight, E., "The DRAGON Computer System: An Early Overview," NATO Advanced Study Institute on Microarchitecture of VLSI Computers, Urbino, Italy, (July 1984).

[MONT77] Montgomery, W. A., "Measurements of Sharing in Multics," Proceedings of the 6th SOSP, Operating Systems Review, V 11, N1 (November 1977).

[NORT82] Norton, R. L., J. L. Abraham, "Using Write Back Cache to Improve Performance of Multiuser Multiprocessors," Intl. Conf. on Parallel Processing, (1982).

[PAPA84] Papamarcos, M. S., J. H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Caches," Proc. 11th Ann. Symp. on Computer Architecture, Ann Arbor, MI, (June 1984).

[RAVI83] Ravishankar, C. V., J. Goodman, "Cache Implementation For Multiple Processors," IEEE Spring Compcon Conference, San Francisco, (Feb 1983).

[RUDO84] Rudolph, L., Z. Segall, "Dynamic Decentralized Cache Schemes for MIMD Parallel Processors," Proc. 11th Ann. Symp. on Computer Architecture, Ann Arbor, MI, (June 1984).

[SMIT82] Smith, A. J., "Cache Memories," *ACM Computing Surveys,* V 14, N 3, (September 1982).

[THAC82] Thacker, C. P., Presentation at UC Berkeley Systems Seminar, (1982).