February 1991

UILU-ENG-91-2211
CRHC-91-7

Center for Reliable and High-Performance Computing

A FEATURE TAXONOMY AND SURVEY OF SYNCHRONIZATION PRIMITIVE IMPLEMENTATIONS

Andy Glew
Wen-mei Hwu

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY none 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE none	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-91-2211 CRHC91-7	5. MONITORING ORGANIZATION REPORT NUMBER(S) none
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL *(If applicable)* N/A	7a. NAME OF MONITORING ORGANIZATION NCR Corporation
6c. ADDRESS *(City, State, and ZIP Code)* 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS *(City, State, and ZIP Code)* Personal Computer Division - Clemson 1150 Anderson Dr. Liberty, SC 29657
8a. NAME OF FUNDING / SPONSORING ORGANIZATION same as 7a. 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* same as 7b.	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
A Feature Taxonomy and Survey of Synchronization Primitive Implementations

12. PERSONAL AUTHOR(S)
Glew, Andy and Hwu, Wen-mei

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Technical	FROM ______ TO ______	February 1991	51

16. SUPPLEMENTARY NOTATION
none

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	multiprocessor, synchronization, computer architecture, parallel processing

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Computer systems that contain multiple intelligent processors require mechanisms for synchronization between these processors. This includes synchronization between a single Central Processing Unit (CPU) and intelligent Input/Output (I/O) units, as well as synchronization between multiple CPUs in multiprocessor systems.

This report considers the implementation details of synchronization primitives. Of the several different types of synchronization, mutual exclusion (locking) is emphasized, because hardware support for this is more common, and other forms of synchronization can be built on top of mutual exclusion. Only the low level question of how synchronization is performed is discussed, not the higher level questions of when and where synchronization should be performed.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE *(Include Area Code)* / 22c. OFFICE SYMBOL

19.

We are concerned with the systems of moderate price and performance that make up most of the computer marketplace. Shared memory systems with cache consistency protocols using a shared bus or a simple interconnect such as a crossbar switch are considered. Figure 1.1 illustrates such a generic multiprocessor system. Multistage interconnection networks are not considered except in passing, because such architectures are costly and will be used mainly by high-performance supercomputers and specialized processors in the next decade. Distributed memory and message passing systems are not considered because, although possibly in the modest price range, the mechanisms investigated in this research do not apply.

The rest of this report is structured as follows: Chapter 2 presents a taxonomy of synchronization features. It defines terms for future use. Chapter 3 surveys other work, particularly proposed cache and bus protocols for synchronization. Chapter 3 also surveys the implementation of synchronization primitives on a wide number of actual implementations, both academic and commercial, and conveys a sense of the present state of the art. Chapter 2 makes forward references to systems in Chapter 3 that demonstrate features of the taxonomy.

These chapters were originally introductory material to an MS thesis [29]. The survey was undertaken to see if the idea proposed by this thesis, the bus abandonment lock, was original. The taxonomy was developed to organize the information in the thesis. The full thesis contains an earlier version of the taxonomy and survey, a description of the bus abandonment lock, simulation experiments, and observations of the synchronization behavior of real systems.

We hope that the taxonomy of Chapter 2 will be useful to others, although it has been our experience that most taxonomies are seldom used by anyone except their author. We have, therefore, emphasized descriptions of features rather than a global classification of systems.

A Feature Taxonomy and Survey of Synchronization Primitive Implementations

Andy Glew and Wen-Mei Hwu
Center for Reliable and High-Performance Computing,
Department of Electrical and Computer Engineering,
University of Illinois at Urbana-Champaign,
1101 W. Springfield, Urbana, IL 61801
`a-glew@@uiuc.edu`

December 14, 1990

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computer systems that contain multiple intelligent processors require mechanisms for synchronization between these processors. This includes synchronization between a single Central Processing Unit (CPU) and intelligent Input/Output (I/O) units, as well as synchronization between multiple CPUs in multiprocessor systems.

This report considers the implementation details of synchronization primitives. Of the several different types of synchronization, ***mutual exclusion*** (***locking***) is emphasized, because hardware support for this is more common, and other forms of synchronization can be built on top of mutual exclusion. Only the low level question of *how* synchronization is performed is discussed, not the higher level questions of *when* and *where* synchronization should be performed.

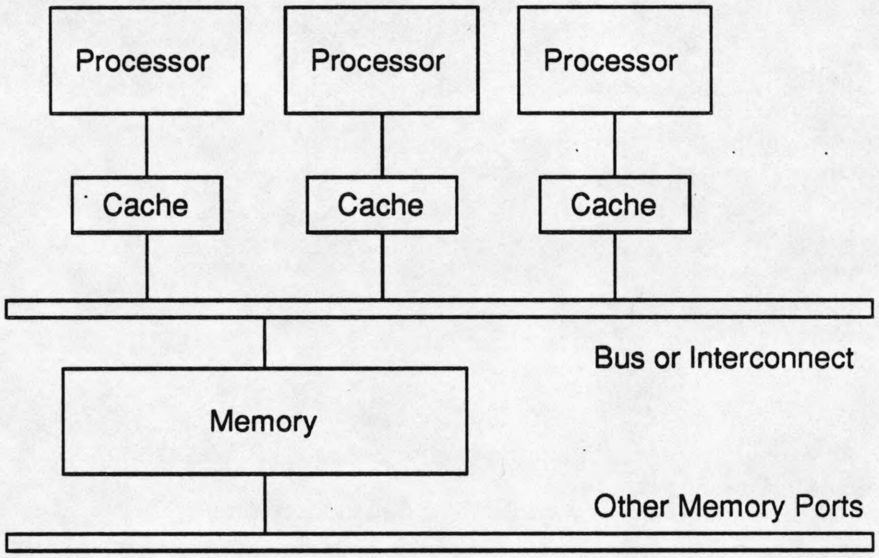

Figure 1.1: Generic multiprocessor system.

We are concerned with the systems of moderate price and performance that make up most of the computer marketplace. *Shared memory* systems with *cache consistency* protocols using a *shared bus* or a simple interconnect such as a *crossbar switch* are considered. Figure 1.1 illustrates such a generic multiprocessor system. *Multistage interconnection networks* are not considered except in passing, because such architectures are costly and will be used mainly by high-performance supercomputers and specialized processors in the next decade. *Distributed memory* and *message passing* systems are not considered because, although possibly in the modest price range, the mechanisms investigated in this research do not apply.

The rest of this report is structured as follows: Chapter 2 presents a taxonomy of synchronization features. It defines terms for future use. Chapter 3 surveys other work, particularly proposed cache and bus protocols for synchronization. Chapter 3 also surveys the implementation of synchronization primitives on a wide number of actual implementations, both academic and commercial, and conveys a sense of the present state of the art. Chapter 2 makes forward references to systems in Chapter 3 that demonstrate features of the taxonomy.

These chapters were originally introductory material to an MS thesis [29]. The survey was undertaken to see if the idea proposed by this thesis, the bus abandonment lock, was original. The taxonomy was developed to organize the information in the thesis. The full thesis contains an earlier version of the taxonomy and survey, a description of the bus abandonment lock, simulation experiments, and observations of the synchronization behaviour of real systems.

We hope that the taxonomy of Chapter 2 will be useful to others, although it has been our experience that most taxonomies are seldom used by anyone except their author. We have, therefore, emphasized descriptions of features rather than a global classification of systems.

Chapter 2

Taxonomy

2.1 Introduction

Synchronization methods may be divided into two classes: those that require special hardware support in the form of atomic Read-Modify-Write (RMW) operations and those that do not. The former are our chief concern, since they are far more common; the latter will be discussed briefly in the next section.

2.2 Synchronization Without Atomic RMWs

Synchronization between a single producer and a single consumer process is trivial, using only simple reads and writes. A number of other synchronization techniques similarly rely on single-writer variables and monotonicity for applications such as enforcing data dependencies between concurrent loop iterations [71].

The earliest algorithm for mutual exclusion of two contending processes requiring only ordered atomic read and write operations is attributed to Dekker by Dijkstra [17]. Dijkstra [16] generalizes this to N processes. Knuth [48] points out that, although Dijkstra's algorithm ensures that all processes are not simultaneously blocked, it is still possible that an individual process may be blocked forever. Knuth suggests a modification that guarantees that a process will eventually enter the critical section, although it may have to wait while other processes perform the critical section $O(2^N)$ "turns." The worst-case waiting time is reduced to $O(N^2)$ turns by de Bruijn [12] and $O(N)$ turns by Eisenberg and McGuire [22]. Peterson [58] presents simpler versions of Dekker's 2-process and Dijkstra's N-process algorithms. Lamport's [50] "Bakery" algorithm has a worst-case waiting time of $O(N)$ turns and is more tolerant of process failures in the critical section. Raynal surveys these and several other synchronization algorithms in his book [59]. More recently, Woo [70] explains how to recycle timestamps in Lamport's algorithm.

All of these algorithms require $O(N)$ storage and $O(N)$ time to acquire a lock without contention. They are unwieldy, and most textbooks quickly assume that atomic RMWs are provided in hardware, and move on to higher level issues such as the implementation of semaphores, monitors, and other concurrency structures. The remainder of this report, however, considers the implementation of atomic RMWs for synchronization in detail.

2.3 Instruction Set

There are two basic synchronization operations that an instruction set must support: (1) acquiring a lock, and (2) releasing a lock. A third operation is frequently used in test-and-test-and-set style spinlocks, such as are discussed in Section 2.5.2: (3) testing a lock. Other operations that may be supported in the instruction set, such as barrier operations, or parallel loop control are much less common and will not be considered.

2.3.1 Releasing and testing a lock

Of these three operations, only acquiring a lock requires an atomic RMW operation. Two issues arise with respect to releasing and testing a lock: (a) Are special instructions provided for releasing and testing a lock, or are normal memory operations used? (b) Is releasing a lock needlessly symmetric with respect to acquiring a lock?

Although releasing a lock and testing a lock are semantically equivalent to ordinary memory operations, their performance characteristics are quite different. Caches, for example, take advantage of locality: a processor is more likely to access a variable that it has used recently than is a different processor that has not used the variable. A synchronization variable, however, is quite likely to be accessed immediately by a different processor [29]. Similarly, the desired memory system behaviour of testing a lock may be very different from normal reads. Section 3.3.5 shows how the use of a normal instruction to test a lock could dramatically decrease performance in an existing system. Special instructions for releasing and testing a lock provide hints to the memory system that may improve performance.

Section 2.6.2.2 shows how a distinct instruction for lock release is desirable in a system with split RMW bus transactions, so that lock release can be distinguished from ordinary memory writes and subjected to the memory interlock that ensures atomicity. Moreover, as coherency models (Section 2.7) evolve, a special instruction for releasing a lock may assume a new semantic role. A widely used model ensures memory consistency only at synchronization operations; it may be desirable to have the choice of ensuring memory consistency either at lock acquisition or at lock release. Some architectures, such as the VAX, prohibit mixing interlocked and normal memory accesses to the same location; others, such as the MVME141, permit intermixing, but may lose cache consistency for these locations.

Some systems have provided instructions for releasing a lock that are symmetric with respect to the instructions for acquiring a lock. This is overkill: lock release does not require an atomic operation, and using an unnecessary atomic operation may exact a performance penalty (Section 3.3.5).

2.3.2 Acquiring a lock

Fine distinctions between atomic RMW instructions are useful because of important differences in implementation complexity. Figure 2.1 shows such a classification, with complexity increasing from top to bottom.

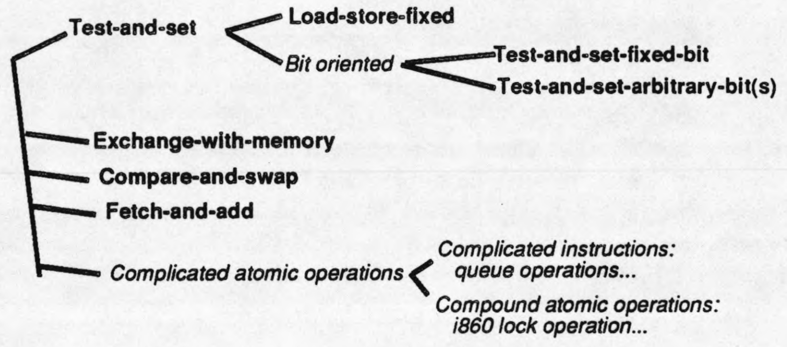
Test-and-set
Load-store-fixed
Bit oriented
Test-and-set-fixed-bit
Test-and-set-arbitrary-bit(s)
Exchange-with-memory
Compare-and-swap
Fetch-and-add
Complicated atomic operations
Complicated instructions: queue operations...
Compound atomic operations: i860 lock operation...

Figure 2.1: Instructions to acquire a lock.

Section 2.3.2.1 discusses the simple instruction set primitives. Sections 2.3.2.2, 2.3.2.3, and 2.3.2.4 describe the last two items, the problem of building up complicated atomic operations. Appendix A proposes a mechanism to solve this problem.

2.3.2.1 Simple synchronization primitives

Let us examine the synchronization primitives provided by a processor instruction set. RMW primitives such as *test-and-set* are the most common. Test-and-set's semantics are presented in Figure 2.2.

```
test-and-set(lock):
  * temp ← lock
  * lock ← 1
    return temp
```

Figure 2.2: Test-and-set.

Atomicity is provided for the operations marked *, usually by the processor blocking interrupts locally and asserting a LOCK signal externally throughout the read and write operations required.

Minor differences in the details of test-and-set have an important effect on the implementation. Some test-and-set instructions modify arbitrary individual bits within a word in memory. These are called *test-and-set-arbitrary-bit(s)*. Since few processors and busses let bits be individually addressed, this requires one bus operation to read the word containing the bits to be changed, a processor operation that changes the bits, and another bus operation to write the bits back. This prevents the implied write bus operations of Section 2.6.2.3 from being used. Some test-and-set instructions address only a fixed bit of each word, typically the most- or least-significant bit. These *test-and-set-fixed-bit* operations are more easily used with out-of-order bus operations and snooping caches.

Exchange-with-memory is more powerful than test-and set, writing an arbitrary value to memory, but exchange-with-memory is often used in a stylized manner where the values exchanged are only 0 and 1. Graunke and Thakkar's queue lock [33] uses the full power of exchange. It is convenient to write the smallest size location that can be separately addressed by a write partial bus transaction.[1] Taking away the ability to write an arbitrary value yields a test-and-set operation that addresses a byte or a word rather than a bit. This *load–store-fixed*[2] operation can be implemented so that the write does not travel across the bus (Section 2.6.2). Of all the simple synchronization operations, load–store-fixed leads to the simplest advanced implementation, in terms of out-of-order bus protocols for synchronization and cache snooping.

Compare-and-swap is a primitive that can "short-circuit" after testing the value of the synchronization variable. Its semantics are described in Figure 2.3.

```
compare-and-swap(loc,old,new)
  * temp ← loc
  * if temp = old
  *     then loc ← new
    return temp
```

Figure 2.3: Compare-and-swap.

This version is as implemented in the IBM 370. It can be used to add an element to a queue, failing if other processors are performing the same operation simultaneously, or in a number of other optimistic concurrency control approaches. Herlihy [36] shows that compare-and-swap can be used to build nonblocking and wait-free concurrent data structures, and that none of the other primitives discussed in this section are

[1] For example, a 64-bit bus may have byte lane signals that permit individual 8-bit bytes to be written.

[2] The differing dash lengths in load–store-fixed are deliberately designed to distinguish between the separate actions of loading the old value of the lock, and storing a fixed value.

of equal power. If the compare operand is always zero, giving ***compare-fixed-and-swap***, fewer bus operations are required in advanced implementations. Compare-fixed-and-swap may be used to write the identity of the lock holder to the lock.

Fetch-and-add is a more complicated operation than test-and-set or compare-and-swap, as shown by Figure 2.4. Fetch-and-add may be implemented in a serial manner, like any other RMW operation. However, fetch-and-add operations may be ***combined***, in the interconnection network of large computer systems such as the NYU Ultracomputer [31] or IBM RP3, or on a bus, as in the Alliant FX. Several processors can receive unique loop indexes, for example, simultaneously. Fetch-and-add can be generalized for arbitrary fetch-and-op operations; indeed, all the other simple synchronization primitives may be thought of as special cases of fetch-and-op [31].

```
fetch-and-add(loc,inc)
* temp ← loc
* loc ← loc+inc
  return inc
```

Figure 2.4: Fetch-and-add.

2.3.2.2 The waiting for a nonrunning process problem

Sometimes it is necessary to perform complicated instruction sequences atomically. For example, if a processor is adding an element to a list in shared memory, other processors should not access the list until all of the pointers have been changed.

In other computer applications, complex operations are built up out of simpler instructions. This may not be practical for complex atomic actions. The RMW primitives described above might be used to acquire and release a lock, as depicted in Figure 2.5, which implements a fetch-and-add operation using mutual exclusion. However, if a user process is preempted or suffers a page fault while in the critical section, the lock may be held for a very long time. Other processes spinning to acquire the lock waste resources uselessly. This is the *waiting for a nonrunning process* problem. With fair-share and preemptive scheduling, this is only a performance problem, not a correctness problem, but in other situations deadlock may result.

```
acquire lock
load r1 ← loc
add r1 ← r1 + inc
store r1 → loc
release lock
```

Figure 2.5: Fetch-and-add implemented using a lock.

This is not a multiprocessor problem. Processes running on different processors can always use a software locking protocol to provide exclusive access to a data structure. The problem is caused by context switching on the local processor. There would not be a problem if the user process could lock down resources such as virtual memory, preventing a page fault in the middle of the critical section, and if the user could block interrupts, preventing preemption by the operating system. Many systems provide limited facilities to lock down resources, such as `plock()` [64]. However, their use usually requires software "root" privilege. Interrupt blocking usually requires hardware privilege. If any user can block interrupts, then a multiuser system can be hung by a malicious or error-prone user with an infinite loop in a critical section. The problem here is not giving the user the ability to block interrupts, but placing a time limit on it.

2.3.2.3 Complicated atomic instructions

Some processors provide complicated atomic operations as single instructions. The Motorola 680x0's CAS2 instruction performs a compare-and-swap operation on two locations at once, and can be used for doubly linked list operations. However, many systems using the Motorola chips do not implement the bus protocol required to make CAS2 atomic, and provide only the more primitive TAS test-and-set operation.

Digital's VAX architecture provides atomic, noninterruptible, operations that insert at the head and tail of doubly linked queues. Since the amount of work in performing these operations is fixed, interrupts cannot be blocked for an arbitrary amount of time. The queues can be addressed relatively as well as absolutely, so that they can be used for synchronization between processes that do not have the same address mapping, or between processes and drivers.

The Gould NP1 also provided atomic queue operations. In addition to the queue operations that required a fixed amount of work, there was an operation to insert in a priority ordered queue that might potentially require scanning all of the elements of the queue. This instruction was removed in subsequent versions of the architecture. It was faster to emulate the queue operations in macrocode inside the kernel, but since the user could not execute the privileged operations required to block interrupts, and since system call overhead would have drowned out the time advantage of macrocode, the atomic queue operations were provided to the user.

Operating System (OS) intervention may be required to perform complicated atomic actions, on systems that do not provide sufficiently high level synchronization instructions in hardware. The BBN TC2000 [6], a large switch-based Non-Uniform Memory Access (NUMA) machine, for example, has only the exchange-with-memory `xmem` primitive available in the instruction set. The TC2000's nX UNIX-based operating system provides the following operations as system calls: fetch-and-add, fetch-and-and, fetch-and-or, compare-and-swap, an atomic clear-then-add operation that can be used for fetch-and-add of a bitfield, as well as atomic operations to find the leading 0 or 1 bit and complement it.

Complicated atomic operations as primitives, whether provided by hardware as instructions or by software as system calls, however, are always limited. The user cannot add to the set of primitive atomic operations. The next section discusses mechanisms that let the user build arbitrary atomic operations.

2.3.2.4 Compound atomic operations

Many systems, such as the AMD 29K, let software explicitly control the lock signal on the bus by writing to a control register. In combination with interrupt blocking, arbitrary instruction sequences can be made atomic, but this usually requires hardware privilege.

A more general solution is provided by the Intel i860. Its `lock` instruction asserts a signal on the external interface, and blocks interrupts, for up to 32 instructions. In other words, interrupt blocking is unprivileged and time limited. Reasonably complicated atomic operations can be coded entirely within this 32 instruction critical section. The critical section must be coded as a transaction with a commit phase, doing no writes before all possible places where an exception can occur have been passed. The operating system can detect when a page fault occurs within such a `lock` critical section, and will restart after a fault at the `lock` instruction. After a page fault, the operating system will ensure that the faulted-in page is made resident, and that other resources required by the critical section are available, in order to guarantee forward progress. This operating system support may require the operating system to interpret the instructions of the critical section to determine what resources are required.[3] The complexity is comparable to that of ensuring that all the pages necessary for address translation of a complicated instruction like VAX MOVC3, on a system with paged page tables, are resident.

Gang scheduling avoids the waiting for a nonrunning process problem by swapping out all threads of a parallel process if any one of them faults. This is wasteful: if there are N processes, one which has been

[3] Stylized code and user-provided hints may be used — for example, no more than 4 pages may be touched by such a transaction, and registers R0..R3 contain addresses in these pages. Compiler support may be necessary to implement these conventions.

preempted and another which is waiting for a resource held by the the preempted process, the remaining $N-2$ processes could be accomplishing useful work which will be delayed if the entire gang of N processes is swapped out. A compromise for a throughput oriented operating system would be to leave the remaining $N-1$ processes running, including the spinning process (whose identity the operating system is not aware of) if there are no processes which are not in the gang to run; but if there are $M < N-1$ other processes runnable, then only $N-M$ processes of the gang will run. In other words, partial gangs are permitted to run only on an otherwise idle system. Random chance determines whether the spinning process is part of the $N-1-M$ process partial gang. A better approach is to tell the operating system which processes are spinning on a lock and are good candidates for preemption, and which processes hold locks and should be preempted only as a last resort. Appendix A sketches such a proposal.

2.4 Memory

The next two sections discuss aspects of synchronization memory. Section 2.4.1 discusses the architectural issue of whether the synchronization memory is special, or indistinguishable from normal memory. Section 2.4.2 discusses the implementation issue of whether the synchronization memory is distributed or centralized. The first is an architectural issue because it affects the programming model visible to human programmers or high level language compilers, although the decision has important consequences for implementation cost and performance. The second is an implementation issue because it affects only the cost or performance of one of the synchronization memory models from Section 2.4.1.

2.4.1 Special versus normal

Synchronization schemes may be classified according to whether they use special synchronization memory or normal memory (Figures 2.6, 2.7, 2.8, and 2.9).

Distinct special memory: Synchronization variables are placed in registers which must be accessed by special instructions. Examples include the Alliant FX [63], the Cray X-MP [39], and the Sequent Balance's SLIC gates [68].

Mapped special memory: Synchronization is done in special memory, which can be mapped into user address space and accessed via ordinary load and store instructions which have special, atomic, side-effects. Examples include the Sequent Balance ALM [68].

Normal memory: Synchronization variables are placed in normal memory locations, but are accessed via special atomic operations. This is the most prevalent.

Tagged normal memory: Some systems use locks that are associated with normal memory locations, but are hidden in tags and are not normal data. Examples include HEP full/empty bits [13], and IBM 801 storage [10].

These different types of synchronization memories raise several issues:

- Special synchronization instructions.
- Protection of synchronization memory.
- Number of synchronization variables.

Special synchronization instructions: Synchronization in distinct special memory and normal memory requires special instructions. Programmers in high level languages may not be able to access these instructions directly, and may be forced to code in assembly when explicitly using synchronization. Parallelizing compilers may be able to use these instructions implicitly. Normal compiled code can be used to access locations in mapped special synchronization memory, but code generators must be made aware of side-effects from accesses to synchronization variables, using mechanisms such as C's `volatile` directive.

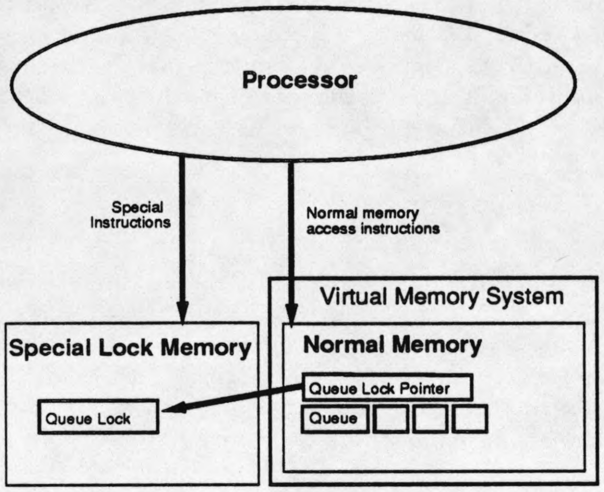

Figure 2.6: Locks in special lock memory.

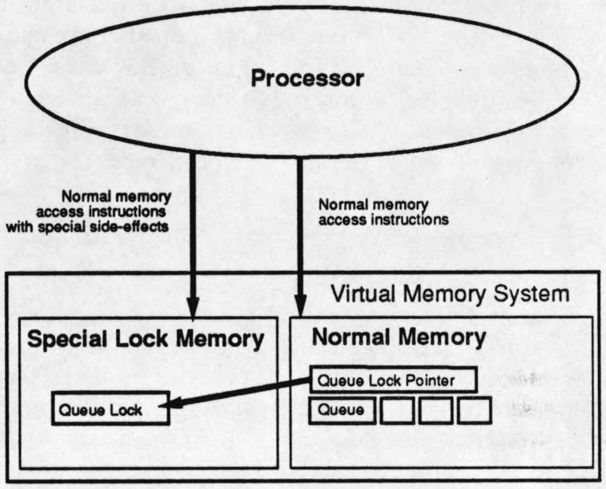

Figure 2.7: Locks in memory mapped special lock memory.

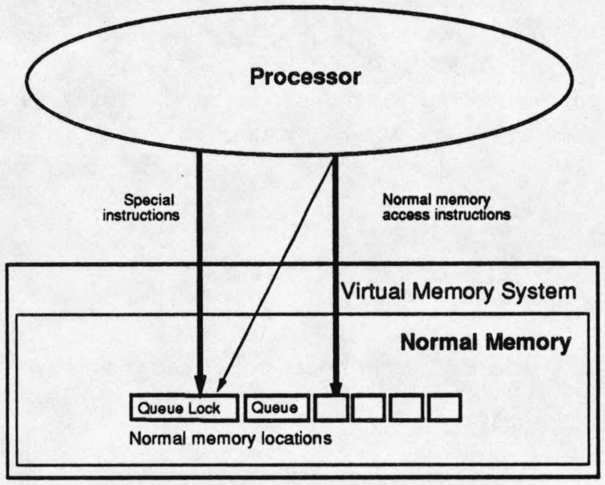

Figure 2.8: Locks in normal memory.

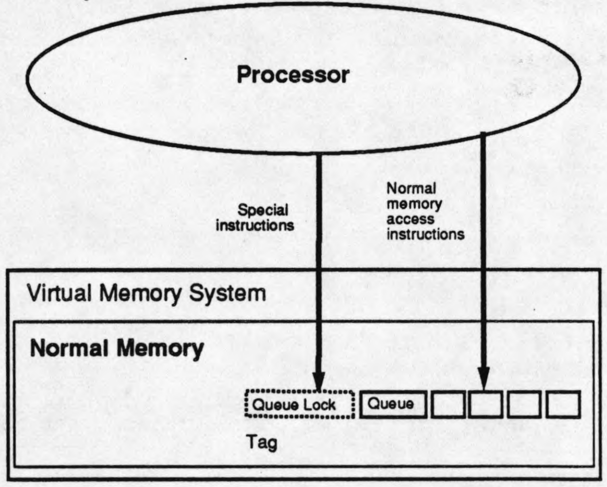

Figure 2.9: Locks in tagged normal memory.

Protection of synchronization memory: Distinct special synchronization memory must either be restricted to privileged users, or some mechanism must be provided to prevent nonprivileged users of the special purpose synchronization store from interfering with each other. Virtual memory mechanisms already provide protection for mapped and normal synchronization memory.

Number of synchronization variables: Synchronization variables in special memory, mapped or unmapped, must be allocated separately from user data structures in normal memory. The association between user data structures and their synchronization variables must somehow be made, by convention, address arithmetic, or pointers (as indicated in Figures 2.6 and 2.7). Usually, however, special synchronization memory provides only a limited number of locks, many fewer than the number of memory locations. Libraries may be provided to multiplex a large number of virtual locks on a small number of hardware locks [62]. Synchronization techniques based on normal memory have no arbitrary resource restrictions (other than the fundamental one of running out of memory).

2.4.2 Distributed versus centralized

Memory used for synchronization may be **distributed** or **centralized**. Distributed synchronization memory maintains a local copy of all the synchronization variables at each processor, broadcasting changes at one processor to all other processors. No single memory is privileged with respect to the others. Centralized synchronization memory has a central "master copy" or primary source for each synchronization variable. Usually, this is of the form of a single physical resource, but there may be multiple physical memories for synchronization, perhaps closer to one processor than another. Also, a centralized memory may have multiple cached copies. Nonetheless, a synchronization variable's "home" is in only one of these memories.

Distributed synchronization memory implementations are usually associated with distinct special synchronization memory or mapped special synchronization memory, as described in Section 2.4.1. Distributed synchronization memory implementations are also usually associated with special synchronization busses which perform the broadcast updates. Figure 2.10 depicts such a system with a special, distributed, synchronization memory. Examples of such systems include Silicon Graphic's MIPS-based machines, Alliant's FX/80, and the Sequent Balance. In addition to the distributed SLIC locks, which could be used only by the operating system, the Sequent Balance also contained atomic lock memory (ALM), a centralized special memory-mapped synchronization memory accessed across the normal memory bus, as depicted in Figure 2.11.

2.5 Caches

2.5.1 Protocols

There are several ways of integrating synchronization operations with caches.

Noncached: Place synchronization variables in noncached memory.

Bypass-cache: Allow synchronization variables to be placed in cached memory, but bypass the cache for synchronization primitives.

Exclusive: Perform the atomic operation on exclusive data in the local cache.

Shared-Abandoning: Initiate a write to a shared unlocked lock variable, but abandon the write if necessary.

The *noncached* approach is simplest, but has the disadvantages of special synchronization memory. The *bypass-cache* approach is probably the most common, as the survey in the next chapter shows. Bypass-cache may invalidate or update cached copies of the synchronization variables in the local or remote caches. The Motorola 88000 bypasses cache and invalidates all cached copies of the synchronization variable. The ARM bypasses cache and updates the locally cached copy of the synchronization variable.

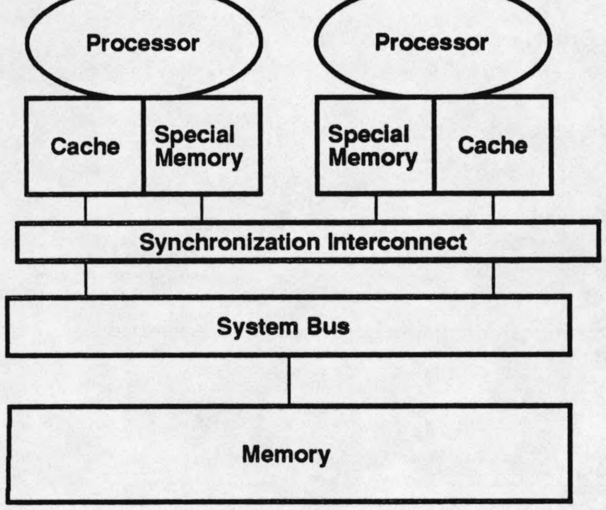

Figure 2.10: Special synchronization memory, distributed copies next to each processor, updated across a special synchronization interconnect (e.g., Sequent SLIC).

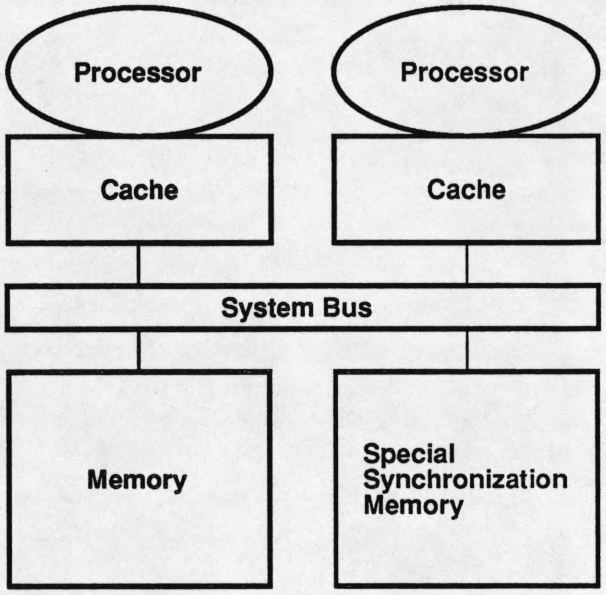

Figure 2.11: Special synchronization memory, one centralized copy, accessed across the normal memory bus (e.g., Sequent ALM).

The *exclusive* approach is increasingly common in ownership-based caches. If the synchronization variable is exclusive within the cache of the processor executing the RMW operation, no bus traffic is necessary. If not, a read-invalidate bus transaction makes it exclusive. Other processors are prevented from acquiring the exclusive cached value of the lock while the RMW operation is being performed in the local cache. The Sequent Symmetry [33] and recent models of the Encore Multimax use the *exclusive* approach.

The *shared-abandoning* approach is an adaptation of the bus abandonment principle to lock operations in snoopy caches. Bus abandonment is discussed in detail in the thesis [29]. If the synchronization variable is exclusive within the local cache, the atomic operation is performed as in the *exclusive* approach. If the synchronization variable is not in the local cache, then a read-invalidate and RMW can be performed as in the *exclusive* approach; or, a normal read leaving the data in shared cache state can be done. If the synchronization variable is shared, then the processor looks at the value of the synchronization variable. If the synchronization variable is not already locked, then the processor initiates a write of the locked value — but the processor must be ready to *abandon* this write if some other processor acquires the lock first.

Note that the bus abandonment principle can be applied to any cached synchronization operations, including the *exclusive* lock protocol, in order to reduce the burst of bus transactions on lock transfer.

2.5.2 Spinloops

The excess bus traffic produced by a naive test-and-set spinloop, used to acquire a lock variable used for mutual exclusion, is well known. A test-and-set spinloop is depicted in Figure 2.12, using an exchange-with-memory operation such as that provided by the Motorola 88100 [56]. This processor uses the bypass-cache strategy, so every cycle around the loop from [1.1] to [1.3] produces an RMW bus transaction. Or, if the exclusive cache strategy is used, the lock variable constantly ping-pongs from one cache to another.

To reduce this bus traffic, Rudolph and Segall [60] proposed the *test-and-test-and-set* operation, which tests a cached copy of the lock variable, avoiding unnecessary bus transactions if the lock cannot be acquired. We find it useful to distinguish between the ideal *test-and-test-and-set* operation, and the actual

```
[1.0] spinlock:
[1.1]   mov     r2,#1
[1.2]   xmem    r2,lock_location
[1.3]   bcnd    ne0,r2,spinlock
[1.4]   <critical section>
[1.5]   st      #0,lock_location
```

Figure 2.12: Test-and-set spinloop.

`test-and-test-and-set` code. In Figure 2.13 a normal memory read is used to test a cached copy of the synchronization variable, although, as we point out in Section 3.3.5 a special instruction for testing the lock may be necessary.

```
[2.0] spinlock:
[2.1]   ld      r2,lock_location
[2.2]   bcnd    ne0,r2,spinlock
[2.3] entry:  used if low contention
[2.4]   mov     r2,#1
[2.5]   xmem    r2,lock_location
[2.6]   bcnd    ne0,spinlock
[2.7]   <critical section>
[2.8]   st      #0,lock_location
```

Figure 2.13: Test-and-test-and-set spinloop.

Anderson et al. [5] point out that **`test-and-test-and-set`** eliminates the steady-state spinloop bus traffic, but suffers a burst of bus transactions on a lock transfer. A *lock transfer* is the activity that occurs when a lock is released, while N other processors are spinning on the lock, until one of the waiting processors has acquired the lock, and the other $N-1$ processors have settled into a steady state. A lock transfer is $1/N$th of "sequentially satisfying N simultaneous requests for a semaphore" [30]. This burst of bus transactions occurs because there is a race from the test- at [2.1] to the atomic test-and-set at [2.5]. When the lock is released by a write operation, all the spinning processors may fall through [2.2]. This race is exacerbated by test-and-set instructions that invalidate all cached copies of the lock even if they do not change the lock's value, and by invalidation-based cache-consistency schemes that require N bus cycles to broadcast a value to N waiting processors (read-combining (snarfing) reduces this in some cache protocols). We have provided a detailed analysis of the bus traffic that occurs in a lock transfer in our earlier paper [28] and thesis [29]. We also show that one hardware feature, abandoning the pending RMW request if another processor gets there first, eliminates the burst. We call this the *bus abandonment lock*.

There is some confusion about the worst-case size of the burst of bus transactions on a lock transfer. For example, Graunke and Thakkar remark that test-and-test-and-set is $O(N)$ while test-and-set is $O(N^2)$. In fact, the distinction is not between test-and-set and test-and-test-and-set, but between different bus scheduling policies for the same synchronization algorithms as we demonstrated in the thesis [29]. The size of the burst of bus traffic is $O(N)$ for typical busses. A number of bus scheduling policies, however, can provide $O(N^2)$ bursts. In particular, a fixed priority bus scheduling policy, where the lock release is broadcast (revalidated), or where reads and writes can be snarfed, produces an $O(N^2)$ lock transfer burst. Pure First In First Out (FIFO) and pure fixed priority scheduling are $O(N)$, not $O(N^2)$. If the size of the burst is larger than the critical section, however, the $O(N^2)$ fixed priority bus scheduling policies reach a saturation point and can, in fact, be better than the $O(N)$ FIFO bus scheduling. Since other bus scheduling

```
Init:   private myPlace
        flags[0] ← 1
        flags[1..P-1] ← 0
        queueLast ← 0
Lock:   myPlace ← fetch-and-add(queueLast)
        while( flags[myplace mod P] = 0 ) spin
        flags[myPlace mod P] ← 0
Unlock: flags[(myPlace + 1) mod P] ← = 1
```

Figure 2.14: Anderson's fetch-and-add software queue spinlock algorithm.

```
lock:   rS ← (myid << 1) | lock_location[myid]
        exchange-with-memory( rD ← mem[lockaddr], mem[lockaddr] ← rS )
        processorAhead ← rD >>1
        meansLocked ← rD & 1
        while( lock_location[processorAhead] = meansLocked ) spin
        lock acquired
unlock: lock_location[myid] ← ~lock_location[myid]
```

Figure 2.15: Graunke's exchange-with-memory software queue spinlock algorithm.

policies can induce this $O(N^2)$ behaviour, bus scheduling policies proposed to solve other problems must be examined carefully.

To reduce the size of the burst of bus traffic on lock transfer, Anderson et al. [5] and Agarwal and Cherian [2] proposed spinlock algorithms with Ethernet style backoff delays. Graunke and Thakkar [33], however, showed that spinlock algorithms with delays still suffer a burst of bus traffic that increases with the number of processors, albeit more slowly.

A queue of spinning processors would reduce the bus traffic on a lock transfer to $O(1)$. Several authors, including Lee and Ramachandran [51] have proposed cache protocols that maintain a queue of spinning processors in hardware. Anderson [4] devised a software algorithm using fetch-and-add to maintain a queue of spinning processors. Each arriving processor obtains a unique sequence number using fetch-and-add. When a processor finished with the lock it notifies the processor with the next highest sequence number. The lock transfer does not require an atomic RMW operation, although a special lock release instruction may be necessary, as pointed out in Section 2.3. The fetch-and-add software queue spinlock algorithm is presented in Figure 2.14, after Anderson [4].

Graunke and Thakkar [33] present a software queueing spinlock algorithm based on the more common exchange-with-memory primitive. One again, each processor has a private lock location. The global lock contains a pointer to the lock location of the last processor on the queue and the value that indicates that that processor has not yet finished with the lock. The key feature of the algorithm is that the pointer and the value can be placed in one packet for manipulation by atomic exchange-with-memory (Graunke and Thakkar make the pointer a processor number, and the value a single bit, fitting both into a single byte). The pointer and value are exchanged with the global lock, and the processor spins on the lock location of the processor ahead of it until the value mismatches. The lock is released by writing any value other than the lock value to the processor's lock location. The value that indicates that that processor has not yet finished with the lock alternates to avoid a race condition. This algorithm is presented in Figure 2.15.

To sum up: Naive test-and-set spinloops constantly produce excess bus traffic. Test-and-test-and-set spinloops eliminate the constant bus traffic, but produce a burst of bus traffic on a lock transfer. The bus abandonment lock is a hardware technique for eliminating this burst of bus traffic. Adding delays to test-and-test-and-set spinloops reduces bus traffic, but the bus traffic still increases with the number of processors. Several hardware cache protocols that maintain a queue of waiting processors have been proposed. Finally, several software queue spinlock algorithms that eliminate the burst of bus transactions have been proposed, that require hardware support that is already present in many systems.

2.6 Interconnects

Interconnect issues for synchronization include specialized interconnect for synchronization, provision of lock signals for mutual exclusion and specialized transactions for synchronization, as well as issues such as request scheduling and abandonment of pending requests. Although these issues are equally pertinent to to generalized interconnection networks and simple busses, busses are emphasized because of their familiarity.

2.6.1 Interconnect structure

Synchronization memory may be accessed in two ways: (1) across the normal memory bus, or (2) across a special interconnect for synchronization.

An argument for a separate synchronization interconnect is that it can be tailored to lock traffic, rather than having a combined synchronization and normal memory bus do everything nonoptimally. This is also the argument for special synchronization memory (Section 2.4.1). Special synchronization busses are often associated with special synchronization memories, often distributed as described in Section 2.4.2. Figure 2.10 depicts such a system with a special, distributed, synchronization memory and a special synchronization bus. Examples of such systems include Silicon Graphic's MIPS-based machines, Alliant's FX/80 concurrency control bus, and the Sequent Balance.

Since busses are increasingly tailored to cache line transfers, there is a mismatch between the block-oriented bus protocol and synchronization accesses which are short and scattered. However, the normal memory bus and a special synchronization interconnect must still be tightly coupled, because of the interaction between synchronization and buffering in the memory system. In systems with separate address and data busses, one possibility is to use the address bus for synchronization operations while the data bus is busy performing block transfers.

2.6.2 Bus transactions

This section concentrates on the type and ordering of bus transactions used in RMWsand the implications for snooping cache protocols. The possible bus transactions are classified as indicated in Figure 2.16.

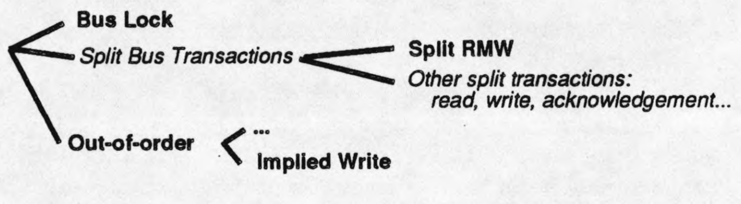

Figure 2.16: Bus transactions for synchronization.

2.6.2.1 Bus lock

The simplest approach is to assert a bus signal that prevents other processors from gaining control of the bus between the read and the write of the atomic operation. This is depicted in Figure 2.17. If the memory boards are multiported, the memory board must lock out the other ports during the locked operation. Bus locking is the most widely used approach in existing systems, and most microprocessors assert an external signal that can be used to control the bus in this way. Locking also applies to more complicated interconnects, such as the Butterfly switch used in the BBN TC2000 [6], in which a locked transaction locks an entire path through the switch.

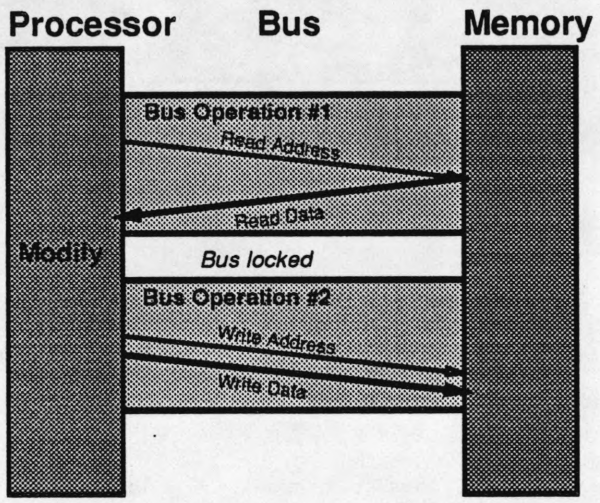

Figure 2.17: Bus transactions — exclusive bus lock.

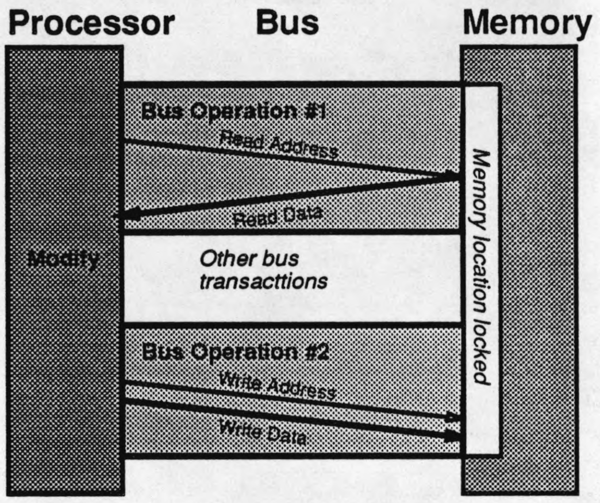

Figure 2.18: Bus transactions — split RMW.

2.6.2.2 Split transactions

Locking the bus throughout the entire operation is wasteful. Overlap can be obtained by splitting the operation, letting other processors use the idle time. Figure 2.18 depicts a split RMW operation, where the read and the write are performed separately. Other processors can use the bus between the read and the write, as indicated in Figure 2.18; use of the bus by other processors during idle times is implied in Figures 2.19 to 2.22.

There are many other possible ways of splitting bus transactions, some of which are depicted in Figures 2.19 and 2.20.

Splitting the read bus transaction into two phases, with the read address presented in the first phase, disconnecting while memory is accessed, and the read data returned in the second phase, is common because memory latency is often much larger than the bus cycle. Similarly, presenting the write address and write data is often split away from the acknowledgement of the write, although Figures 2.18 to 2.22 do not show this.[4] FUTUREbus+ [42] is an example of a system with decoupled transactions. Less common is decoupling of the write address and write data, as depicted in Figure 2.20. The Encore NanoBus [23] is an example of such a system, with address and data busses that are separately arbitrated.

Split bus transactions are a feature distinct from split address and data busses. Split reads can be performed on systems where the address and data busses are multiplexed. Splitting write address and write data makes less sense if the address and data busses are multiplexed, or if they are arbitrated for together.

[4] Some systems use "write-and-forget" in which no acknowledgement is returned from memory as a performance enhancement.

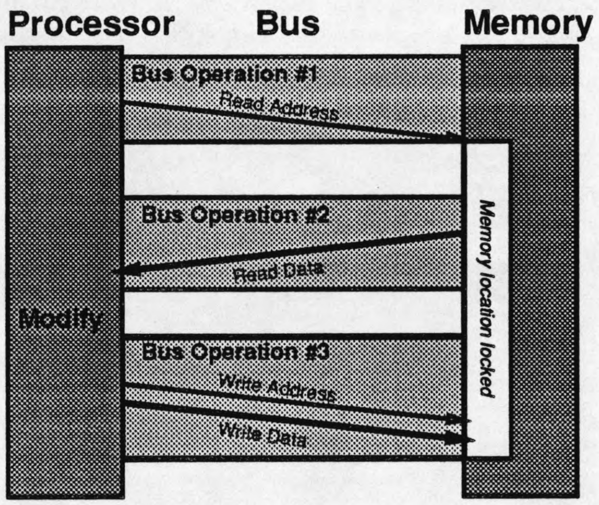

Figure 2.19: Bus transactions — split read.

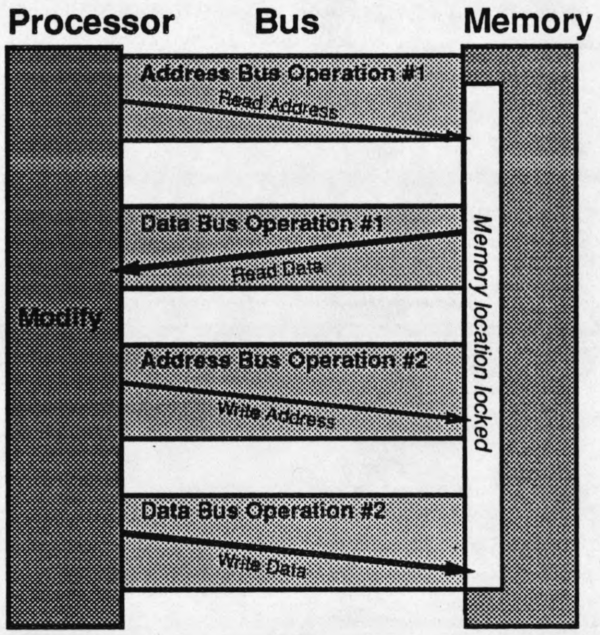

Figure 2.20: Bus transactions — split read and write.

With a split RMW, some mechanism other than a bus lock must ensure atomicity. Typically, this is done by a hardware lock at the memory board. A read-lock bus transaction acquires the hardware lock; a write-unlock bus transaction releases the hardware lock. There is often only one hardware lock per memory board, but the VAX 6200 [15] is an example of a system that supports multiple hardware locks, with a small associative store at the memory boards that permits several atomic operations to be in progress at the same time.

A decision must be made as to what operations will be prevented by the hardware lock: interlocked operations only, or normal noninterlocked operations as well. Permitting noninterlocked operations to proceed can be a performance advantage, particularly when the granularity of the hardware lock is coarse. However, permitting noninterlocked operations to proceed means that a normal memory write must be distinguishable from the memory write that releases a software lock.[5] This is a reason for providing a separate instruction for lock release (see Section 2.3.1).

Operations that cannot obtain a hardware lock are retried later. This can lead to a situation called **interlock lockout**, where one processor's request is repeatedly retried because it happens to access the synchronization variable while another processor holds the hardware lock. Not only can the atomic operation that acquires the lock be blocked, but the operation that releases the lock can also be blocked.

There must be special provision in the bus scheduling mechanism for the write that releases the software lock. Otherwise, the write to release the software lock may arrive when the hardware lock is set by a test-and-set spinloop, resulting in deadlock as the write is constantly retried.

In a personal communication, a Digital Equipment Corporation (DEC) engineer called this the *interlock lockout* problem. It had been discovered on the VAX 6000 line of computers, which implemented hardware

[5]Terminology: software uses synchronization operations applied to a synchronization variable to manipulate a *software lock*. The *hardware lock* is the interlock controlled by the memory system that ensures the atomicity of the synchronization primitives that software employs.

locks at memory with a small associative lock store. It was "solved" by adding an extra level of arbitration[6] invoked after an interlocked operation timed out.

We investigated [29] more elegant mechanisms for avoiding the interlock lockout problem. Giving priority to the bus request that releases the lock removes the problem, but it can also induce the $O(N^2)$ burst of bus transactions on a lock transfer described in Section 2.5.2. A better solution is to modify the meaning of bus retries in a FIFO bus scheduling policy. Rather than retrying, the bus transactions that are excluded by the hardware lock are marked as "blocked," and do not proceed until the bus transaction that releases the hardware lock that they are blocked for is observed on the bus. Rather than being reissued with a priority for FIFO bus scheduling corresponding to the time at which they were unblocked, the unblocked bus transactions resume with their original priority and come to the head of the FIFO queue.

Bus snooping cache hardware can also be used to enforce atomicity. The cache whose processor is in the middle of performing an atomic RMW tells other processors that attempt to access the synchronization variable to retry. This can be done even if the snooping cache protocols of Section 2.5.1 are not used for synchronization. Exclusivity maintained by bus snooping does not propagate across multi-ported memory boards. However, in the special case of producers (CPUs) on one port and consumers (I/O) on another port of a dual ported memory, bus snooping can be used to maintain atomicity on one side so long as the test-and-set RMWs on the other side short circuit without writing memory if they cannot obtain the lock (ie. if they are compare-and-swap).

2.6.2.3 Out-of-order

Advanced busses support synchronization orders where the read and write may be performed out of order. as illustrated in Figure 2.21. The address is transmitted only once, although there are two data transfers. The new data may replace the original value of the synchronization variable, as in an exchange-with-memory operation, or it may be used to modify the original value, as in a fetch-and-add, or test-and-set of bits that cannot be individually addressed across the bus, in which case the new data may be a bit number or mask. FUTUREbus+ [42] contains three such advanced RMW operations: swap, compare-and-swap, and fetch-and-add.

Load–store-fixed operations have a particularly pleasant out-of-order implementation. Since a fixed value (typically 0, or 1, or all 1s) is being written, it does not need to be transmitted across the bus. Only a single use of the address bus and a single use of the data bus are required (Figure 2.22).[7] Early models of the Encore Multimax support such an implied-write operation. Combined with completely decoupled address and data busses, it permits overlap of multiple atomic operations. Fetch-and-add with a fixed increment can also be implemented with an implied-write bus transaction.

Given that out-of-order and implied-write bus transactions can dramatically reduce the amount of bus traffic required to implement synchronization operations, why do more systems not use them? Indeed, there seems to be a trend the other way, with early models of the Encore Multimax using an implied-write bus transaction, while later models make the lock exclusive in the local cache (see Section 2.5). The answer seems to be the extra hardware expense required to implement out-of-order synchronization operations, particularly the extra intelligence at the memory board. Making the lock exclusive in the local cache, by contrast, uses hardware that is already available in a snoopy cache system. Another factor is that synchronization operations are not very frequent in the throughput oriented applications that are the largest part of the market for shared memory multiprocessors like Encore and Sequent. Synchronization variables, also, are not very frequently shared among processors, in which case caching them exclusively provides better performance than optimized bus transactions. Finally, special bus operations for synchronization still suffer a burst of bus traffic on lock transfer, if they are implemented without bus abandonment.

[6] The DEC engineer described it as an extra "half-level" of arbitration using an open collector line.

[7] The whole data bus need not be used to return the old lock value, if it is only ever 0 or 1.

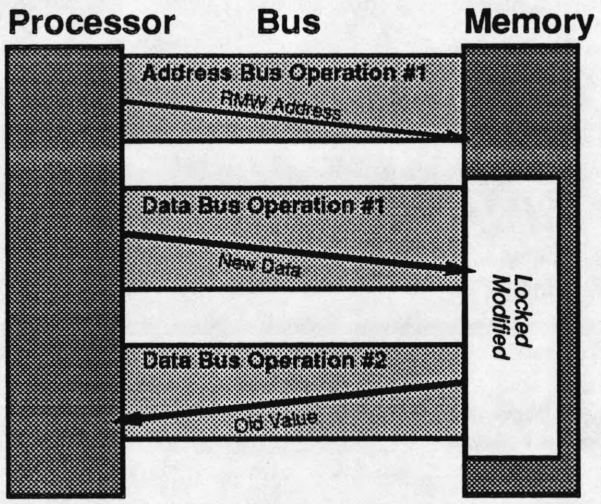

Figure 2.21: Bus transactions — out-of-order RMW.

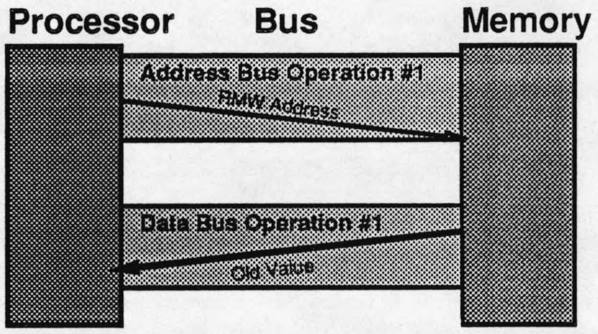

Figure 2.22: Bus transactions — implied write.

2.7 Buffering

Caches are only one aspect of buffering in advanced memory systems. There may also be queues, for example, a queue between the bus interface and the cache, to buffer invalidations or updates from the bus, or a queue of writes that have not yet secured access to the bus. Synchronization operations usually combine an atomic RMW operation with a memory consistency operation such as flushing buffers. Recent research has begun to explore the possibilities of separating these operations.

Dubois, Scheurich, and Briggs [20] provide a good introduction to memory consistency models and define conditions for strong and weak ordering of events. Strong ordering means essentially that every memory transaction reflects a serialized memory model. Weak ordering means that coherent memory state is mandated only after synchronization effects. In practice, this means that synchronization primitives must flush queues.

Buffering can be flushed either at the time a lock is acquired (Gould NP1 semaphore operations), or at the time the lock is released (as in the Sequent Balance SLIC gates). These are *receiver* and *sender* synchronization, respectively. Sender synchronization is a local operation. Receiver synchronization requires the flushing of data in remote buffers (at least those of the last processor to hold the lock), but some of the data may already have been flushed. Sender synchronization, however, may be decoupled, with the local processor continuing to execute from cache while its buffers are clearing. A form of sender synchronization, called *release consistency*, has been proposed for the DASH multiprocessor [25]. It has been shown to be equivalent to weak ordering for parallel programs with sufficient synchronization.

In either case, flushing buffers in the memory system may be expensive. Overlapping with other operations is desirable. Gupta's *fuzzy barrier* [35] is a step in this direction. Instead of a single synchronization point in the code, a barrier region is defined where processes can continue doing useful work while waiting. No processor may leave the barrier region until all processes have entered the barrier region, but execution within the barrier region may continue. We have proposed *split synchronization*: (1) requesting that buffers be flushed early, (2) performing independent operations for which a consistent state is unimportant while flushing occurs, (3) finally waiting until the flushing initiated in 1 is complete [27]. Similarly, Bisiani and Ravishankar [7] split atomic RMWs into separate *issue* and *verify* primitives so that useful work can be performed during the RMW latency of 40 or more cycles in their PLUS system.

Chapter 3

Survey

3.1 Introduction

Chapter 2 classified many different possible features of synchronization implementations. Chapter 2 used real systems as examples to clarify certain points, but the presentation was always from feature to system. The present chapter turns this around and surveys the synchronization features of a variety of systems. The presentation is from system to feature. This perspective is necessary because it conveys the complex interactions of synchronization features in real systems.

This survey gathers together information about synchronization that is scattered across multiple documents. For example, to understand synchronization in a commercial system, it may be necessary to read the following manuals:

Processor Chip Architecture Manual: Defines atomic instructions.

Processor Chip Implementation Manual: May say, for example, that atomic instructions bypass the on-chip cache and assert a LOCK signal on the chip's external interface.

Processor Board Manual: Indicates how the board cache interprets the LOCK signal.

Bus Manual: Describes interlocked bus transactions and locking signals.

Memory Board Manual: Indicates whether the memory board observes locked operations.

Each of these documents may be produced by a different company! System manuals seldom provide a comprehensive overview of synchronization (the BBN TC2000 manual [6] is a notable exception).

Section 3.2 briefly surveys previous academic work on synchronization. Section 2.5.2 and Section 2.7 survey previous work that is more closely related to our research.

The survey of real systems is loosely divided into three parts: Systems (Section 3.3), Microprocessors (Section 3.4), and Busses (Section 3.4). There are many cross-relationships among these sections. For example, the MVME141 board level system (Section 3.3.7) cannot be understood without the MC68030 microprocessor (Section 3.4.7) and the VME bus (Section 3.5.1). Unfortunately, the interrelationships do not form a linear graph suitable for textual presentation. Therefore, we have arbitrarily decided to sort the survey entries alphabetically.

The chief criterion for inclusion of a system in this survey was availability of technical information. It is appropriate, however, to give a broad overview of the contents of the survey.

The author encountered the MVME141 and Gould NP1 systems while employed in industry. The complexities of their support of synchronization piqued the interest that led to this research. The MVME141 is a good example of how microprocessor, memory board, and bus support for synchronization interact, sometimes destructively. The Gould NP1's synchronization facilities show the importance of memory buffering and providing high level synchronization operations to unprivileged users. The original Encore exemplifies the implied-write bus transaction described in Section 2.6.2.3. The Encore, Sequent Symmetry, and

SPUR take advantage of ownership-based caches for the RMWs. The VAX implementations are examples of interlocking at the memory board. The other systems in this survey all have distinctive synchronization support.

The microprocessors surveyed, however, are all very similar. They may support a variety of synchronization instructions, but they nearly all bypass any on-chip cache and assert an external lock signal during the RMW operation. External logic is required to let these microprocessors take advantage of the synchronization support afforded by the standard busses in the last part of the survey.

The chapter concludes with comparative tables summarizing the synchronization features surveyed. Once again, the presentation is from feature to system. Certain conclusions about the "state of the practice" of synchronization can be made from this comparison.

3.2 Papers

Bitar and Despain [8] introduce special states for synchronization to their cache protocol, rather than setting and clearing locks in data memory. The first state indicates that a cache has locked a data block. The second state indicates that other processors are waiting for the lock. The proposed implementation tightly couples bus and cache. Bus arbitration gives priority to waiting processors. A special register snoops for an unlock, interrupting the processor when the the lock is acquired. Additional complexity arises when locked cache lines are replaced, requiring their state to be saved. Bitar and Despain also mention implementing atomic RMW operations without any lock on cache or bus. If the final write generates a miss, an exception is produced indicating that atomicity was violated.

Goodman [30] describes the Wisconsin Multicube's grid of caches and busses and proposes a synchronization operation that sets up a queue of waiting processors. A node executing a SYNC operation allocates space in its local cache for the lock. This space is used to store the id of the next node that requests the lock. On lock release, the lock is forwarded to the first requester waiting in the queue. The queues can break down if a locked cache line is replaced, degrading to test-and-set across several busses. Apart from this the queues provide first-come, first-serve scheduling. This QOSB (queue on synchronization bit) operation is the predecessor of the QOLB (queue on lock bit) operation in the proposed Scalable Coherent Interface (SCI) standard [44],[43].

Dubois and Briggs [19] mention briefly a synchronization scheme that can take advantage of a snooping protocol on a single bus. LOADSYNC loads the lock in the cache, and activates a snooper which monitors the bus. CONDSTORE stores 1 in the lock provided the snooper has not flagged it. This is the synchronization primitive of the MIPS-II architecture. It can be used to implement arbitrary atomic operations. However, it has the same race that leads to the burst of bus transactions on lock transfer that ordinary test-and-set RMWs suffer.

Eggers [21] considers Rudolph and Segall's RWB protocol. She assumes that the policy of first write-broadcast update, second write-broadcast invalidate. A cache protocol which makes the synchronization variable exclusive in the local cache on the first write or RMW access is assumed. Eggers points out that this protocol would not benefit locks, which are modified twice per critical section. She suggests performing broadcasts for the first two writes, followed by invalidation and exclusive access if no other processor intervenes.

Other papers on synchronization are discussed elsewhere in this report. Section 2.5.2 in particular mentions the test-and-test-and-set spinloops which were introduced by Rudolph and Segall [60], analyzed by Anderson et al. [5],[3], and improved on by Graunke and Thakkar's queue lock [33]. Section 2.7 lists some papers in the field of memory consistency models and synchronization.

3.3 Systems

3.3.1 Alliant FX

The Alliant FX/8 and FX/80 [63] contain caches on both Instruction Processors (IPs) and the Computational Elements (CEs) cluster. However, since there is only a single shared cache (with multiple modules) for the computational cluster, CE interactions are not affected by cache consistency. Each CE contains a Concurrency Control Unit (CCU) containing 8 shared registers. The CCUs are connected by a special Concurrency Control Bus (CCB), whose most important signals are listed in Table 3.1.

Table 3.1: Alliant CCB signals.

Bits	Function	Bits	Function
8	ready	1	start
8	idle	1	wait
8	active	1	quit
2	select	33	data

The CCU and CCB implement a variety of synchronization operations, such as advance and await. Alliant's compilers make extensive use of these facilities for synchronization of concurrent loops. Only one of these operations is presented in detail here.

The CCB implements combining for a bus-based fetch-and-add. There is a ready bit for each shared register and an active bit for each CE. To increment a shared counter, a CE turns on its active bit and the ready bit for the shared register it wants to modify. When all CEs agree as to which register to modify (usually there will be no conflict, but occasionally arbitration will be necessary), each active CE increments its copy of the counter by the number of active lines. The old value of the counter, incremented by the CE's position in the set of active lines, is returned to the process running on the CE. This may be used to increment a loop counter. For example, if CEs 1, 4, and 5 each try to increment the same shared register the active mask will be 01001100. If the original register value is 0, the shared register will be set to 0+3 in each CE's CCU. CE 1, 4, and 5 will receive 0+1, 0+2, and 0+3, respectively.

The next generation Alliant FX/800, based on the Intel i860, has no CCB, but emulates some of the FX/8's concurrency control instructions in software. Like the Sequent Symmetry, the Alliant FX/800 is evidence of a trend away from special, separate, synchronization hardware to solutions better suited to stock microprocessors.

3.3.2 BBN TC2000

The BBN TC2000 [6] is a large shared memory parallel computer based on the Motorola MC88000 and an interconnection network called the Butterfly switch. The TC2000 descends from BBN's Butterfly and Monarch parallel computers. Switch paths in the interconnection network are connection-based and bidirectional. Locking is performed for the MC8100's native synchronization instruction, `xmem`, exchange-with-memory, and for memory accesses with a special "augmentation" register bit set. Privilege is required to modify this control register. The `xmem` operation bypasses the cache. Locked accesses set up by the augmentation register do not necessarily bypass the cache, and should be addressed only to uncached memory regions. Locking prevents the switch path used to connect the processor to memory from being used for other purposes. A locked memory board cannot be used by other processors until the locked transaction is complete. If the locked transaction is directed to the VMEbus I/O interface, VMEbus mastership is maintained until the transaction is complete. Bypassing of the lock protocol for 8 megabyte blocks of memory can be controlled with the system memory mapping unit (which provides a level of memory mapping beyond that provided by the MC88200 Cache and Memory Management Unit (CMMU)). Bypassing permits a CPU to access instructions and page tables in its local memory even if the local memory is locked by a remote processor.

The TC2000 technical manual [6] devotes more than a dozen pages to synchronization and locking. It is one of the best manuals we have encountered in this respect. The manual recommends randomized delay in spinlocks to reduce contention. Such backoff is, in fact, provided by the hardware for retrying of all memory accesses that are not immediately able to obtain a switch path connection. The manual points out that a test-and-test-and-set spinloop, which takes advantage of the MC88200's snooping caches by spinning on a cached copy of the lock location, does not apply to the TC2000, because the TC2000's interconnection network defeats snooping. The manual also points out that a location being used by an `xmem` spinlock should not be accessed by normal loads or stores, since the `xmem` is not atomic with respect to these other accesses.

The TC2000's nX UNIX-based operating system provides the following operations as system calls: fetch-and-add, fetch-and-and, fetch-and-or, compare-and-swap, an atomic clear-then-add operation that can be used for fetch-and-add of a bitfield, as well as atomic operations to find the leading 0 or 1 bit and complement it. These are implemented using the privileged control register to lock arbitrary transactions.

The TC2000 is the largest system considered in this survey, and the only system with a switch-based interconnection network rather than a bus. Nonetheless, many of the same principles of synchronization discussed for simpler systems apply.

3.3.3 Cray X-MP

Although the Cray X-MP [39] has no cache, its special synchronization registers are interesting. CPU intercommunication is provided by three clusters of shared registers. A cluster consists of 8 24-bit shared address registers, 8 64-bit shared data registers, and 32 1-bit synchronization registers. These registers are protected, allocated under OS control to all, any, or none of the processors. They are accessible from both user and monitor modes. Deadlock detection is done in hardware.

3.3.4 Encore Multimax

Encore's Multimax [23] is based on the National Semiconductor NS32000 series microprocessor and Encore's proprietary NanoBus. The NanoBus is a 100 Mbyte/s bus consisting of decoupled address bus (32 bits + parity, round-robin scheduling), data bus (64 bits + parity, fixed priority), and interrupt vector bus, with centralized arbitration. Pended operations are tagged with the requestor's identity.

In the original NS32032-based Multimax, up to the 300 series, the lock operation bypasses the cache and produces an implied-write transaction on the bus. Use of the NS32xxx interlocked operations generates a special NanoBus cycle that reads the requested location, returns status to the processor, and, if the lock contains 0, atomically sets it to all 1s. This requires one use of the address portion of the NanoBus and one use of the data portion of the NanoBus to return the data. These bus activities are decoupled. Multiple lock operations may overlap. The setting of the data is done at the memory, and requires no further bus activity. This operation permits locks to be placed anywhere in memory, which Encore has taken advantage of in its user mode parallel libraries.

Note that Encore has taken the test-and-set-arbitrary-bit synchronization operation of the NS32000 family and implemented it on the bus as a load–store-fixed operation. In other words, the semantics of the microprocessor's operation have been changed by the system implementation. The NS32000 has a CBITI clear-bit-interlocked operation which is symmetric with the SBITI set-bit-interlocked operation. Encore's parallel libraries, however, use an ordinary store instruction to release the lock because the memory system cannot distinguish CBIT from SBITI.

The third generation Encore Multimax 500 series, based on the NS32532 processor, does not use the special bus protocol for synchronization. Locks are made exclusive in the local cache. As in the Sequent Symmetry and the Alliant FX/800, this is evidence of a trend away from special support for synchronization.

3.3.5 Gould NP1

Although now obsolete, the Gould NP1 ECL minisupercomputer [32] is interesting because it provides examples of several important issues for synchronization:

- separate instructions to test and release locks
- interaction between synchronization and memory buffering
- complex queue operations

The NP1 was designed to support up to 8 processors in a 4 bus, 2 processor per bus configuration. A uniform shared memory model was provided by an Inter-System Bus Link (ISBL) [69]. The cache consistency protocol was write-through with invalidation.

The synchronization instructions were SSM, ZSM, and TSM. SSM (set semaphore in memory) was used to acquire a lock, ZSM (zero semaphore in memory) was used to release a lock, and TSM (test semaphore in memory) was used to test a lock. A semaphore was any arbitrary bit in memory.

A multiple-bus NP1 system contained a great deal of buffering, both in a local write-through buffer, and in the ISBLs. SSM waited until all queues in the system had been flushed before proceeding. This is *receiver*-based synchronization, since the processor executing the SSM would then proceed to execute a critical section, and needed to ensure that all changes to data in that critical section had been received by the local cache. Originally, the ZSM instruction also waited until all queues were flushed, *sender*-based synchronization, but since locks were always used in a stylized fashion with SSM to lock and ZSM to unlock, the ZSM queue flushing was removed as an unnecessary impediment to performance.

The NP1 cache was accessed both by the CPU and the bus snooper, with a single port for invalidates and data accesses. A 4 deep invalidate queue buffered the snooper from the cache port. In normal operation the CPU had priority access to the cache. The invalidate queue received higher priority access to the cache if there was danger of overflow, or in order to maintain write ordering. Test-and-test-and-set on the NP1 might be naively coded as in Figure 3.1 using the ordinary TBM (test bit in memory) instruction to perform the test in cache. When the lock was released by a remote processor, an invalidation would be queued for the local cache. The repeated TBM requests, however, would take priority over the queued invalidation preventing the change in the lock from being seen. This situation is depicted in Figure 3.2. The TSM instruction was provided for such spinlocks and lowered the processor priority to allow invalidates.

```
test: tbm lock
      bne test
      ssm lock
      bne test
```

Figure 3.1: Gould NP1 naive TBM spinloop.

Atomic queue insertion/deletion instructions were also provided. These were slower than the corresponding sequences of simpler instructions and were therefore not used by the kernel, which implemented complex operations guarded by semaphores. However, the atomic queue instructions provided atomicity with respect to interrupts to user processes, which would otherwise have had to execute much more expensive system calls. In addition to the usual queue operations which insert/delete at the head or tail of a queue, another type of queue instruction inserted into a queue sorted in priority order. This instruction was subsequently removed because of its potentially unbounded delay.

3.3.6 MIPS-based systems

The MIPS R2000 and R3000 have no architectural support for synchronization, but this has not prevented multiprocessors from being built with them.

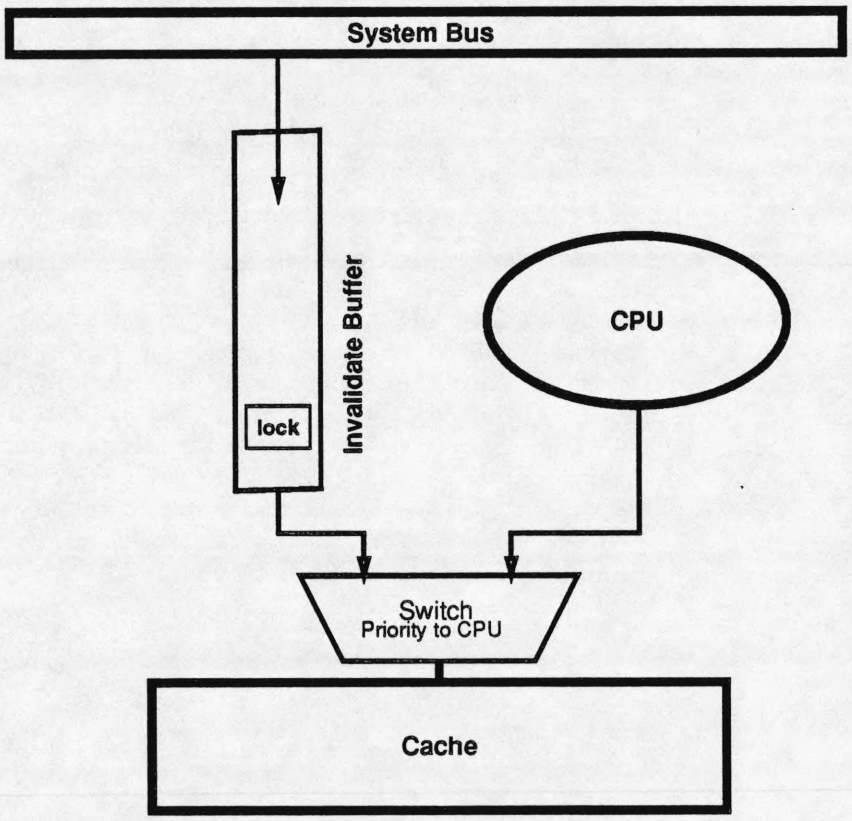

Figure 3.2: Gould NP1, spinlock blocking invalidation.

Silicon Graphics (SGI)'s PowerSeries machines use multiple R2000s and R3000s which provide no atomic instructions. SGI has implemented test-and-set spinlocks with a custom gate-array per processor and a private bus. The gate array is actually a bus-watching cache that maintains the state of 64K spinlocks. The UNIX kernel is allocated 32K, and user processes are allocated 32K, which can be mapped into a process's address space and accessed like memory. An SGI engineer described the advantages of this approach as follows:[1]

> Moving the synchronization traffic to a separate bus has several advantages, the primary reason being that it takes load off the memory bus. Latency and cache coherency protocols on the synchronization bus can be tailored to lock traffic, rather than having one bus design that does everything nonoptimally.

SGI has therefore implemented a special, memory-mapped, distributed, synchronization memory with a special synchronization interconnect.

On the DECStation 3100, a uniprocessor system based on the MIPS processors, the implementors of MACH at Carnegie-Mellon University have described two solutions [24]. The first is to define an extended instruction TAS with an opcode that causes a trap, with test-and-set emulated in the kernel. Since these machines are uniprocessors, the kernel routines need only to block interrupts while performing the test-and-set, and do not need to provide atomicity with respect to the bus. Their second solution, which they describe as not portable, not necessarily working on multiprocessors, uses an algorithm derived from Lamport [50].

3.3.7 MVME141

The Motorola MVME141 system serves as an example of how synchronization primitives may depend on cache, bus and memory configurations. The MVME141, depicted in Figure 3.3, is a MC68030 board with 64K of on-board cache and two bus interfaces: VME, used for memory and I/O, and VSB, typically used only for memory. The cache protocol is write-through with invalidation and write allocation. A bus monitor on the VMEbus performs invalidations. There is no monitor on the VSBbus. Memory traffic can be directed towards the VME or VSB busses based on physical address; or the system can be configured in a VSB bounce mode, where all accesses are attempted on the VSB and passed to the VME in case of failure; or in a VSB read-only mode, where read traffic is directed to the VSB, but write traffic is directed to the VME for bus monitoring.

The MC68030 processor supports TAS, test-and-set, and the more complicated CAS and CAS2, compare and swap, primitives, which bypass the on-chip (non-snooped) cache and assert an RMC signal on the external interface. When RMC is asserted, the board cache is bypassed and memory operations proceed directly to the VMEbus. Since the VMEbus is nonpreemptive, exclusivity on the bus can be obtained by not surrendering bus mastership between transactions. This is sufficient for single-ported memory boards.

The VMEbus standard does not support a LOCK signal line to signal exclusivity with respect to other access ports. TAS, however, involves only a single byte location and appears on the VMEbus as a single address strobe, AS, with two different data strobes, DS, one for the read, the other for the write. Dual-ported VSB/VME memory boards in the system may be configured to treat a continuous AS on the VMEbus as a lock signal. They will permit VSB accesses to interleave between VME accesses only when AS is not asserted. Thus, TAS locks out the VSB, providing atomicity. Unfortunately, without this convention VSB memory cycles can begin earlier, providing greater system throughput. It penalizes all memory accesses, not just TAS cycles.

CAS and CAS2 may involve more than one memory location and do not maintain the same address on the bus throughout their access. They are not atomic unless a reserved pin on the VMEbus is used as a LOCK signal, and the memory boards are strapped appropriately.

The VSB bus supports a LOCK signal, but as mentioned above, the MVME141 does not have a VSB bus monitor. This would not cause a problem for the atomic RMC instructions, since they bypass the cache,

[1] Paraphrased from a USEnet posting [18]

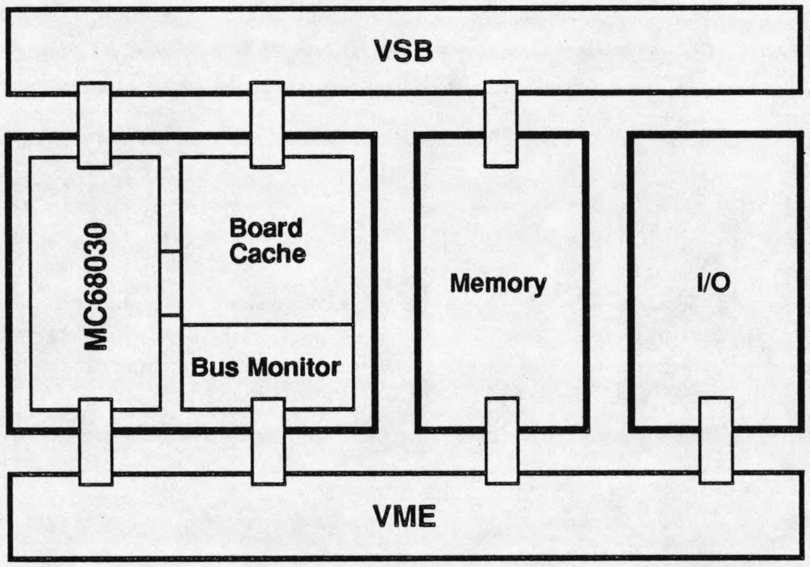

Figure 3.3: MVME141 system.

but attempts to read those locations by normal memory accesses might yield inconsistent data. In any case, synchronization with I/O devices on the VME must be possible.

The MVME141 is an example of a system in which some of the RMW operations defined by the processor may or may not be atomic, depending on board, bus interface, and memory configurations. It demonstrates that there is sometimes a performance cost associated with supporting synchronization.

3.3.8 Pyramid

This RISC minicomputer's interlocked instructions bypass cache. Its interlocked instructions are

BITSW Bit test and set word interlocked: if all the bits of word in memory under a 32-bit mask are 0s, the bits under the mask are all set and a condition code set.

ICMPW Increment and compare word interlocked.

Interlocked bit clear and decrement operations are provided symmetrically. Exchange with memory is not interlocked. The most notable thing about these instructions is that they manipulate multiple bits at the same time and thus they can access multiple locks in the same operation.

3.3.9 Sequent

Sequent Computer Systems has successfully developed and marketed two generations of multiprocessor systems.

The first generation, Balance [68], based on the NS32000 series microprocessor, has an invalidation-based cache. Balance contains a special purpose System Link and Interconnect (SLIC) serial bus, usable only by the kernel. This allows interrupt and synchronization traffic to be separated from the main memory bus. Each SLIC chip implements 64 binary semaphores, called gates, and commands to atomically test and set them. The SLIC uses its local copy of the gate value to determine if SLIC-bus messages are necessary. The SLIC waits until the processor write buffer has emptied before unlocking a gate.

Balance also contains Atomic Lock Memory (ALM), 64K of locks on each Multibus memory adaptor, with the property that any read request atomically sets the location to 1 as a side effect. These can be mapped into user memory. The Balance Parallel Programming Manual [62] suggested ways of implementing cached shadow copies of locks so as not to produce excessive bus traffic by spinning on the uncached atomic lock memory. It also suggested techniques to multiplex several logical locks behind a single software lock. However, the overhead of accessing ALM is sufficiently high that applications on Balance may use spin-locking based on `xchg`, the exchange-with-memory instruction supplied by the processor [5].

The second generation Symmetry [53] has copy-back caches. The SLIC is no longer used for synchronization; new, faster transparent parallel locks, such as described by Graunke and Thakker [33], are used. Both the kernel and the users use the same type of locks for mutual exclusion. Once again, there seems to be a trend away from special purpose support for synchronization, as evidenced by the Sequent Symmetry, Alliant FX/800, and Encore Multimax 500.

3.3.10 SPUR

The SPUR processor [67],[26], whose name is an acronym for Symbolic Processing Using RISC, is a member of the Berkeley family of processors that includes RISC I, RISC II, and SOAR. SPUR is notable for its support of tagged data, trapping if an instruction is executed with operands of unexpected type. A SPUR system consists of a modest number of custom processor/cache boards on a modified NuBus [40] with standard, unintelligent, NuBus memory boards. SPUR has a dual system bus: (1) cache-to-cache; (2) cache to memory and I/O. The cache-to-cache bus supports the Berkeley protocol, with transfers between caches that do not update memory.

The instruction set synchronization primitive is TEST_AND_SET, which is the same as an LD_32 read operation except that it also performs an atomic read-modify-write which writes a 1 in the cache. SPUR does not use NuBus's arbitration-based locking. Instead, SPUR uses the exclusive cache protocol for synchronization, performing TEST_AND_SET within the cache, on a word brought into the cache with a `read-for-ownership` transaction on the SPURbus. Thus, SPUR's TEST_AND_SET does not produce any bus transactions if the word is already private in the local cache.

3.3.11 VAX

Digital Equipment Corporation's VAX is one of the most successful computer architectures in history, and has had many different implementations. First, this section presents the VAX architectural support for synchronization. Then, two implementations are discussed.

The VAX architecture [52] provides seven atomic operations:

Test-and-set-arbitrary-bit: BBSSI and BBCCI (branch on bit clear and clear interlocked)

Fetch-and-add: ADAWI

Queue instructions: INSQUE, REMQUE, INSQHI, INSQTI, REMQHI, REMQTI

BBCI (branch on bit set and set interlocked) and BBCCI (branch on bit clear and clear interlocked) are conventional test-and-set-arbitrary-bit RMWs. ADAWI (add aligned word interlocked) is a fetch-and-add operation. Its memory operand is constrained to be a well-aligned word, an unusual limitation for a VAX instruction. This architectural constraint means that implementations do not need to worry about fetch-and-add operands that overlap bus boundaries.

There are two classes of queue operations on the VAX. The first class, absolute queues, contains instructions INSQUE (insert entry in queue) and REMQUE (remove entry from queue). Since pointers in these doubly linked queues are absolute addresses, processes communicating via these queues must have the queues mapped at exactly the same location in their virtual address space. The absolute queue instructions are noninterruptible and can be used to communicate between processes running on the same processor, or between a process and an interrupt service routine running on the same processor. They are not, however, interlocked, and cannot be be used to communicate between processes running on different processors, or with I/O devices.

The second class, relative queues, contains instructions INSQHI (insert entry into queue at head interlocked), REMQHI (remove entry from queue at head interlocked), INSQTI (insert entry into queue at tail interlocked), and REMQTI (remove entry from queue at tail interlocked). Since pointers in these queues are displacements relative to the current queue entry, processes communicating via relative queues need not share address space. This permits relative queues to be used with I/O devices which lack virtual memory mapping hardware. The relative queue instructions are both noninterruptible and interlocked and can be used to communicate between processes on different processors. Bit 0 of the queue header is used as a semaphore. Bit 0 is set via an atomic RMW when the queue is being manipulated, reducing the length of time the hardware interlock is maintained. Consequently, while the absolute queue operations can synthesize head and tail operations by changing an offset, the relative queue operations must be provided in separate head and tail varieties. Relative queues, unlike absolute queues, must be quadword aligned, an architectural feature which simplifies implementation.

The VAX architecture [52] contains the following statement:

> Once an interlocked operation has occurred, other processors and I/O devices are locked out of performing interlocked operations on the same control variable until the interlock is released.

Normal read and write operations, however, are not excluded by the interlock. Normal writes must not be used to release a lock that is being manipulated by an interlocked RMW— they may disappear. Moreover, in multiprocessor systems, access to shared variables must be interlocked by software executing one of the interlocked instructions. In other words, normal memory operations and interlocked operations cannot be intermixed, and memory consistency is guaranteed only after an interlocked operation.

The VAX-11/780 [14] implements the interlocked operations with separate **`interlock read masked`** and **`interlock write masked`** SBI (Synchronous Backplane Interconnect) bus transactions. The SBI commander (processor) asserts the SBI interlock signal during an **`interlock read masked`** command. The addressed memory controller continues to maintain the interlock signal until it receives an **`interlock write masked`**. All memory controllers respond with busy to **`interlock read masked`** commands while the SBI signal is asserted. In the terminology of Chapter 2, this is a split RMW bus transaction, with exclusion maintained by a single hardware lock at memory.

In the VAX 6200 [15] the interlocked instructions bypass the cache. They are mapped to **`interlock read`** and **`write unlock`** operations on the XMI (Extended Memory Interconnect) bus. Interlocked reads set a lock flag associated with the location. Interlock reads to a location with its lock flag set result in the responder returning a "locked" response. The MS62A memory module contains 16 interlock flag registers. These registers contain a 24-bit address which is compared to the address of interlocked reads in order to maintain exclusion. Unlock writes clear the lock flag. A transaction identifier in the interlock flag registers detects whether an unlock write is initiated by the correct processor. In brief, VAX 6200 interlocked operations bypass the cache and use split RMW bus transactions with a hardware lock at memory of 16 byte granularity for up to 16 locks.

3.4 Microprocessors

3.4.1 Acorn RISC Machine

The Acorn RISC Machine (ARM) swap instruction (SWP) performs the actions depicted in Figure 3.4.[2] This is an exchange-with-memory operation with the advantage that the source operand is not destroyed. ARM3 SWP bypasses the write-through on-chip cache without flushing or invalidating. If the addressed memory is cached, the cache is updated with the `Rc` data as the word is written to memory. The read always goes off chip. ARM3 asserts a LOCK signal at the external interface.

```
memory[Ra] → temp
Rc → memory[Ra]
temp → Rb
```

Figure 3.4: ARM swap instruction.

3.4.2 AMD 29000

This RISC microprocessor [1] contains no on-chip cache. The instruction to acquire a lock is LOADSET, a load–store-fixed operation with the LOCK* signal asserted on the external bus. In addition, LOADL and STOREL operations are provided. which implement read and write operations with the LOCK* signal asserted, not full RMWs. These can be used as the distinct instructions that test and release a lock. LOADL may be used by a system designer to implement a one-cycle implied-write synchronization operation (see Section 2.6.2.3), with appropriate support from the memory system. The LK (Lock) bit in the Current Processor Status register permits software control of the LOCK* bus signal for complicated instruction sequences, but requires hardware privilege.

[2]The information in this entry was obtained by personal communication with an Acorn engineer.

3.4.3 IBM RS/6000

The IBM RS/6000 [34] contains one of the more unusual and innovative synchronization mechanisms. The "database memory" developed in the IBM 801 project and described by Chang and Mergen [10], which was first partially implemented commercially in the IBM RT ROMP storage system [37], has been more fully implemented in the RS/6000 [9]. Hardware data locking and granting are combined with page protection in the Translation Lookaside Buffer (TLB). A *Special* bit in a segment register enables the data locking mechanism for pages in that segment. A lock bit and a 16-bit Transaction ID are associated with each 128 byte line of a page. Lock type bits in the segment register distinguish read and write locking. Contention, indicated by a mismatch of the Transaction ID, causes a trap to a software lock manager.

This scheme, therefore, is synchronization in tagged normal memory. Ordinary read and write instructions are used to access database memory if there is no contention, but special instructions are used by the lock manager to manipulate the Transaction IDs if there is contention. Lock release is performed as part of a database commit operation.

Clearly this scheme is efficient only if the amount of lock contention is low. Otherwise, trap overhead would reduce performance. These papers describe only uniprocessor implementations, in which synchronization is performed between multiple processes on the same machine. It is unclear how this scheme operates in a multiprocessor environment. A conventional load-store-fixed instruction is provided.

3.4.4 Intel 80x86

The Intel 80x86 family, which includes the 8088, 8086, 80186, 80286, 80386/i386, and i486 microprocessors,[3] is characterized by a LOCK prefix instruction, which makes the instruction following the prefix indivisible by blocking interrupts and asserting a LOCK signal.

On the uncached and nonvirtual memory 80286, the LOCK prefix could be applied to multiple cycle instructions, such as repeated string operations.

On the Intel i486 [45], with virtual memory, the LOCK prefix is more restricted. It cannot be used to lock string instructions, due to the difficulty of ensuring that all the pages touched by such an operation are in memory simultaneously. Nonetheless, the LOCK prefix can be applied to a wide variety of instructions:

- BIT TEST and SET/RESET/COMPLEMENT
- XCHG register and memory
- ADD, OR, ADC, SBB, AND, SUB, XOR between memory and register/immediate
- NOT, NEG, INC, DEC to memory

The LOCK prefix produces an exception if applied to an instruction that does not read-modify-write memory.

Exchange-with-memory is the preferred synchronization primitive, since the LOCK# bus signal is asserted for any memory-based XCHG instruction without requiring the LOCK prefix. The i486 also contains two new synchronization instructions, XADD and CMPXCHG. XADD implements fetch-and-add. CMPXCHG is a compare-and-swap that checks in the on-chip cache before committing to an external RMW cycle, as indicated by timings of 7 cycles if register and memory are equal, 10 cycles if unequal, +2 cycles if cache miss.

The i486's on-chip cache can be disabled or made write-through on a page-by-page basis. Copy-back caching is not implemented. LOCKed operations bypass the cache. Write buffers, which can permit a read to pass buffered writes, are flushed before an RMW instruction is executed on the bus (*sender*-based synchronization).

[3] The numbering scheme for this processor family has recently been changed.

3.4.5 Intel i860

The Intel i860 [46],[49] does not snoop for its on-chip cache. Atomic operations are provided by setting the BL bit in the privileged **dirbase** register, which asserts the LOCK# signal on the bus. The LOCK# signal can also be set by the LOCK instruction. This begins an interlocked sequence on the next data access that misses the cache, setting the BL bit. Interrupts are disabled, and the bus is locked until explicitly unlocked. The critical section must be coded as a transaction with a commit phase, doing no writes before all possible places where an exception can occur have been passed. The operating system can detect when a page fault occurs within such a LOCK critical section and will restart after a fault at the LOCK instruction. The UNLOCK instruction unlocks the bus on the first data access that misses the cache after it. The LOCK and UNLOCK instructions are executable from user mode. A 32-instruction counter prevents holding the lock too long.

3.4.6 MIPS

The MIPS R2000 and R3000 processors [47], like their academic relatives, the Stanford MIPS and MIPS-X processors [11],, have no synchronization operations. Nonetheless, several multiprocessor systems have been implemented around them (see Section 3.3.6).

The synchronization operations of the second generation MIPS-II architecture, implemented in the R6000 processor, are *load locked* and *store linked.* Load locked performs a load and sets a lock in a status register. The MIPS R6020 bus interface chip provides one external status register bit per memory board. Store conditional completes the store operation only if no other store conditional has accessed the linked memory location since the load linked instruction. Store conditional returns 0 if successful, 1 if not, in a general purpose register destination.

In other words, the MIPS-II architecture has exposed the components of the atomic RMW operation, permitting software to synthesize arbitrary operations. For example, a fetch-and-add is coded as follows:

```
L:      LL      T1,(T0)          ; read semaphore          ADD      T2,T1,1          ; increment semaphoı
```

This approach, however, only works with the exclusive cache protocol or hardware locks at memory, and is incompatible with special out-of-order bus transactions for synchronization.

3.4.7 Motorola 680x0

The MC68030's [54] chief synchronization primitive is TAS, a test-and-set-fixed-bit operation which sets bit 7 of a byte in memory. The CAS instruction performs a compare-and-swap. CAS2 performs a similar operation on two locations simultaneously. CAS2 is useful in building list data structures. Several MC68030 systems such as the MVME141 (Section 3.3.7) have memory systems that implement TAS atomically, but not CAS or CAS2.

The MC68030 contains an on-chip cache, but does not snoop. TAS, CAS, and CAS2 bypass the on-chip cache and assert the RMC* signal externally.[4]

The MC68040 asserts the signal LOCK during TAS, CAS, and CAS2. A LOCKE signal indicates the end of a LOCKed sequence, separating back-to-back LOCKed sequences. The next bus grant is provided when LOCKE is asserted. The bus arbiter has been moved off-chip on the MC68040, and the external arbiter may allow the bus to be taken away even while LOCK is asserted. When control is returned, the processor assumes that the other bus master has not interfered with its atomic operation. In other words, the external bus master may choose not to lock the bus, especially if it can determine by other means that the intervening transactions do not disrupt atomicity. TLB entries for the LOCKed operands are preloaded.

[4]RMC* is Motorola's designation of the lock signal. The * indicates that the signal is asserted low. Similarly, LOCK# is the lock signal on some Intel 80x86 processors.

LOCKed instructions bypass the cache. These instructions write back matching dirty entries and invalidate all matching entries.

3.4.8 Motorola 88000

The first implementation of this Reduced Instruction Set Computer (RISC) architecture consists of the MC88100 CPU [56] and the MC88200 Cache and Memory Management Unit (CMMU) [57]. The CMMU supports uncached, write-through, and copy-back pages. The synchronization primitive is `xmem`, exchange-with-memory, which bypasses the cache, asserts a LOCK signal on the external M-bus, and leaves invalid cache entries for the lock location behind.

3.4.9 National Semiconductor 32xxx

The NS32000 [38],[61] series' synchronization primitives are SBITI (set-bit-interlocked), CBITI (clear-bit-interlocked), and TBITI (test-bit-interlocked). These assert the ILO interlocked operation pin on the microprocessor interface. CBITI is, however, symmetric with SBITI, and lock release is therefore performed by an ordinary store instruction on the Encore Multimax, a system which uses the NS32000 series.

3.4.10 SPARC

SUN SPARC's [65] synchronization primitives are LDSTUB and SWAP. LDSTUB is an atomic load–store-fixed of an unsigned byte, which reads a byte in memory and writes all 1s. SWAP atomically exchanges memory and a register. Privileged LDSTUBA and SWAPA versions are provided to access an alternate address space such as I/O. Memory remains unchanged if one of these instructions traps, but the register value may or may not have changed. The Fujitsu M86900 implementation asserts a lock signal on the chip's external interface.

3.5 Busses

It is difficult to separate a processor from its bus, particularly for proprietary systems that use neither standard microprocessors nor standard bus architectures. However, the support provided for synchronization by several widely available bus standards can be discussed.

3.5.1 VME

VME [41], the mostly widely used industrial standard bus, has little support for synchronization. Since the bus protocol is nonpreemptive, a bus master can perform arbitrarily complicated sequences of operations simply by refusing to surrender the bus between transactions. This is insufficient, however, to provide resource locks for multiported boards.

As mentioned in Section 3.3.7, some systems use the continuous assertion of the address strobe bus signal as an interlock. This limits atomic RMWs to single words and cannot support complicated atomic actions such as queue operations. Some systems use a reserved pin on the VME connector as a lock signal, but this is nonstandard. VME's limitations for synchronization have been remedied in VME's related and descendant busses, VSB, FUTUREbus, and FUTUREbus+ which all provide lock signals.

3.5.2 VSB

VSB stands for VME Subsystem Bus. This bus is usually an auxiliary bus on the P2 connector of VME boards, providing a fast path to memory, free of I/O interference. Among the VSB control signals are CACHE, indicating the cacheability of data, and LOCK*, used to signal indivisible access. The VSB specification [55] warns that the LOCK* signal can cause system deadlocks. It classifies indivisible sequences into two types:

INTRASEQ An *Intradomain Indivisible Sequence* is one in which access to all byte locations involved is controlled by the same arbitration mechanism.

INTERSEQ An *Interdomain Indivisible Sequence* is one in which access to the byte locations involved is controlled by two or more arbitration mechanisms.

An INTERSEQ might, for example, involve locations on both the VSB and VME busses in a dual-bus system. The LOCK* signal is sufficient for implementing INTRASEQs, but higher level protocols are required for INTERSEQs.

3.5.3 NuBus

NuBus has a simple distributed arbitration mechanism. NuBus distinguishes between a bus lock and a resource lock.

Bus locking relies on the nonpreemptive bus scheduling mechanism. NuBus arbitration begins with a START cycle, in which processors that desire the bus, called contenders, drive arbitration codes onto several wired-logic lines. The contenders back off, stop driving their arbitration signals, based on what is observed. Two cycles after the start a winner is determined. After a winner has used and released the bus, another arbitration contest is performed. No new contender may join in an arbitration contest until all prior contenders have withdrawn. The set of processors that participates in the initial arbitration defines a *wave*. Usually a master will withdraw after using the bus. However, if the current master does not withdraw, since no new contender may join the current wave and the priorities used in the arbitration remain fixed during a wave, then the current master will win again.

Resource locking is done by issuing an attention-bus-lock cycle as the first transaction of a bus tenure locked as described above. All bus modules with resources that may be locked, e.g., multiport boards, must monitor the bus. If the module is addressed after an attention-bus-lock-cycle then it shall lock any other ports out. The resource lock is released by an attention-null cycle. This permits arbitrary sequences of operations to be locked.

3.5.4 FUTUREbus

FUTUREbus supports a variety of cache consistency protocols on the same bus [66]. FUTUREbus uses a distributed arbitration protocol similar to NuBus. FUTUREbus is not preemptive. For synchronization with modules that have multiple ports, indivisible operations are signalled by an LK flag during the connection phase of every transaction. Multiple slaves may be locked by performing an address-only transaction, with LK* asserted, with each slave in turn. LK* must remain asserted during any subsequent address transfers. This provides read locked, write locked, read partial locked, and write partial locked bus transactions.

If any of the responding agents returns `Busy`, the requester must abort all locked transactions in progress and try again later. A slave may return `Wait` rather than `Busy`, if the slave will itself free the resource. Pending locked operations need not be aborted, but may time out.

3.5.5 FUTUREbus+

FUTUREbus+ [42] is an extension to FUTUREbus. Like FUTUREbus, FUTUREbus+ supports a variety of cache protocols simultaneously. There is also a message passing protocol.

FUTUREbus+ contains several "advanced" locked operations, out-of-order:

Swap Locked: Master writes new data to slave. Master reads old data from slave. Slave writes new data to location.

Fetch and Add Locked: Master writes add operand. Master reads old data from slave. Slave adds add operand to location.

Compare and Swap Locked: Master writes two operands to slave, compare operand and swap operand. Master reads old data from slave. Slave compares compare operand with old data and if equal writes swap operand in location.

All of these locked operations may be *connected*, with read and write transactions as described above; or they may be *split*, in which case the master ends its bus tenure, and at a later time the responding agent, who was originally the slave, performs a write transaction to return the data to the requester. These locked operations may not, however, be combined with FUTUREbus+'s copy back cache protocol.

3.5.6 Other busses

As mentioned above, the Encore NanoBus has a special out-of-order load–store-fixed operation. Sequent, Alliant, Cray, and Silicon Graphic's have provided special synchronization busses in their systems.

3.6 Conclusions

Tables 3.2, 3.3, 3.4, 3.5, and 3.6 summarize the results of this survey, and compare the systems surveyed in the framework of the feature taxonomy of Chapter 2. Not every system is represented in every table. Moreover, several systems for which there was not enough information to warrant a textual entry are presented in the tables. The tables are sorted primarily alphabetically although there is some attempt to keep systems near to the microprocessors they employ, except for Table 3.4 which is in approximate chronological order.

Table 3.2 compares the architectural synchronization instructions. There are many test-and-set instructions, a fairly large number of exchange-with-memory instructions, and a variety of other atomic RMWs. Relatively few machines provide separate instructions for releasing and testing a lock.

Tables 3.3 and 3.4 compare bus and interconnect support for synchronization, and Table 3.5 compares the memory models for synchronization variables. Several systems provide special interconnects and memories for synchronization, but there are two examples of product families (Sequent and Alliant) that started with special synchronization support and abandoned it in later models.

Table 3.6 compares cache support for synchronization. The overwhelming majority of systems bypass the cache and lock the bus during synchronization operations. Some recent systems with copy-back caches make synchronization variables exclusive in the local cache before performing an atomic RMW. These systems do not need special bus support for synchronization between processors, although such support may still be necessary for synchronization with I/O devices that do not participate in the cache protocol.

Processor or System	Atomic read-modify-write (RMW) operations — acquiring a lock							Releasing a lock		Testing a lock
	test-and-set	exchange-with-memory	compare-and-swap	fetch-and-add	queue	compound	privilege required	separate instruction	symmetric	separate instruction
Alliant FX				●						
AMD 29000	load–store-fixed (LOADSET)					LK bit	yes	STOREL	no	LOADL
ARM3		**swap**								
AT&T WE32x00		●								
Clipper C300	fixed-bit (TSTS)									
Gould NP1	arbitrary-bit (SSM)				●			ZSM	no	TSM
Intel 80x86	●	XCHG	CMPXCHG	XADD						
Intel i860						BL bit	yes			
						lock	no			
MIPS R2000										
MIPS R6000		LoadLocked/StoreConditional								
MC680x0	fixed-bit (TAS)		CAS		CAS2					
MVME141	fixed-bit (TAS)									
MC88100 MC88200		**xmem**								
BBN TC2000		**xmem**				augmentation register	yes			
NS32000	arbitrary-bit (SBITI)							CBIT	yes	TBITI
Encore Multimax	load–store-fixed (SBITI)							not used		TBITI
SPARC	load–store-fixed (LDSTUB)	SWAP								
SPUR	load–store-fixed (TEST_AND_SET)									
VAX	arbitrary-bit (BBSSI)			ADAWI	●			BBCCI	yes	

Table 3.2: Synchronization instructions.

Table 3.3: Synchronization interconnects — systems.

Processor or System	Special Synchronization Interconnect	Normal Memory Bus	Bus Transactions
Alliant FX/8 Alliant FX/80	CCB		bus-based fetch-and-add combining
Alliant FX/800	removed	●	bus lock
BBN TC2000		●	exclusive lock on switch path and memory board
Cray X-MP	●		
Encore Multimax 320		●	implied write
Encore Multimax 520		●	read for ownership
Gould NP1		●	split RMW
IBM RS/6000		●	
MVME141		●	bus lock
Sequent Balance	SLIC		
		ALM	normal read implied write
		xchg	bus lock
Sequent Symmetry		**xchg**	read for ownership
SGI PowerSeries	●		
SPUR		●	read for ownership
VAX 780		●	split RMW lock all memory
VAX 6200		●	split RMW lock hexword

Table 3.4: Standard bus support for synchronization.

Processor or System	Special Synchronization Interconnect	Normal Memory Bus	Bus Transactions
VME		●	nonpreemptive continuous address strobe reserved LOCK signal
VSB		●	nonpreemptive reserved LOCK signal
NuBus		●	nonpreemptive attention-bus-lock
FUTUREbus		●	locked transactions
FUTUREbus+		●	locked transactions
		●	exchange-with-memory compare-and-swap fetch-and-add

Table 3.5: Synchronization memory — noting the type of memory or the instructions used to access it.

Processor or System	Distinct Special	Mapped Special	Normal	Normal Tagged
Alliant FX/8 Alliant FX/80	CCU registers			
BBN TC2000			**`xmem`**	
Cray X-MP	shared registers			
Denelcor HEP				Full/empty tag bits
Encore Multimax			SBITI	
Gould NP1			SSM	
IBM RS/6000				"database memory"
MVME141			TAS	
Sequent Balance	SLIC gates	ALM	**`xchg`**	
Sequent Symmetry			**`xchg`**	
SGI PowerSeries		64K of spinlocks		
SPUR			TEST_AND_SET	
VAX			interlocked operations	

Table 3.6: Cache protocols for synchronization.

Processor or System	Noncached	Bypass	Exclusive	Shared abandon
Systems				
BBN TC2000	augmentation register	xmem		
Encore Multimax 320		●		
Encore Multimax 520			●	
Gould NP1		●		
MVME141		●		
Sequent Balance	ALM			
		xchg		
Sequent Symmetry			●	
SPUR		●		
VAX 780		●		
VAX 6200		●		
Microprocessors with on-chip cache				
ARM3		● update		
Clipper C300		●		
Intel 80x86		●		
Intel i860		●		
MC68030		●		
MC68040		●		
MC88100 MC88200		● invalidating		

Chapter 4

Summary

Chapter 2 discusses the range of possible features an implementation of synchronization operations might have. Although most taxonomies are doomed to be used only by their creators, we hope that some of the distinctions made in this chapter will prove useful. Presenting all of these concepts together in one place will help future students of synchronization.

Chapter 3 is a broad survey of synchronization implementations past and present. We feel confident that this collection of real machine characteristics will be useful to others. The survey of real machines has already been revealing: frequently colleagues would say "Nobody in his right mind would design a system with X," where X is a particular combination of features from the taxonomy of Chapter 2, only to find that an existing, commercial machine in the survey in fact has X.

We conclude with several opinions concerning trends for future implementations of synchronization and future research.

Lock instructions: All future computer system architectures should provide three distinct synchronization instructions: (1) acquiring a lock, (2) releasing a lock, and (3) testing a lock. Not providing such instructions makes certain implementation approaches impossible, while these instructions can always be made synonymous to normal reads and writes in a particular implementation.

Microprocessors designed to be used in a variety of computer system architectures should provide two different instructions for acquiring a lock: (a) a conventional instruction that produces separate read and write operations at the external chip interface, with an interlock signal asserted, and (b) an instruction that produces only a read operation at the external chip interface, with external signals that permit it to be distinguished from normal reads and the lock testing instruction. The second instruction would be used to build an implied-write synchronization primitive, in cooperation with the memory system.

Synchronization primitives: The following are the most important choices of synchronization primitives for future computer architectures: load–store-fixed, exchange-with-memory, compare-and-swap, and fetch-and-add synchronization primitives.

Of the primitives suitable for implementing conventional locks, load–store-fixed is the most suitable for implementing advanced out-of-order implied-write bus transactions, and the easiest for implementing bus abandonment.

Exchange-with-memory requires more bus transactions in advanced implementations than load–store-fixed and requires more logic to implement bus abandonment. However, exchange-with-memory is desirable because it is required for Graunke and Thakkar's software approach to eliminating the burst of bus transactions on a lock transfer.

Compare-and-swap, exchange-with-memory, and load–store-fixed are all of comparable complexity when implemented with a cache protocol that makes locks exclusive in the local cache. Compare-and-swap has advantages for implementing concurrent data structures.

Fetch-and-add is the most suitable primitive for combining in an interconnection network and, therefore, for massive scientific parallelism. However, the complexity of fetch-and-add combining may not warrant providing it as an instruction for computer architectures that have more modest goals. It is pointless to provide fetch-and-add if the microprocessor chip interface does not make it visible to the memory system.

Compound synchronization: Instructions such as the i860's `lock` instruction, that allow an unprivileged user to block interrupts for a short period of time and combine arbitrary instructions into compound atomic operations, should be provided in all future computer architectures.

Operations such as the MIPS-II architecture's `load-locked` and `store-conditional` instruction, although permitting arbitrary RMWs to be synthesized, do not permit optimized bus transactions to be employed.

Gang scheduling is unnecessary: User compound atomic operations such as described in Appendix A can be used to provide hints to the operating system that can let partial gangs run without the waiting for a nonrunning processor problem. This will provide better use of processor resources, improving throughput as well as individual job latency on a shared system.

The only reason why gang scheduling should persist, and the above operating system scheduling approach not be employed, is if processors become so cheap that nobody cares about wasting them.

Special Synchronization Memories and Interconnects: There seems to be a trend away from special memories and interconnects for synchronization. At least commercial architectures, the Sequent Balance and Symmetry, and the Alliant FX series, have moved towards synchronization in normal memory across the normal bus.

Synchronization = atomic operations + memory consistency: Atomic operations and obtaining a consistent view of memory by flushing buffering are two separate concepts. They are combined in most present-day synchronization operations, but need not be so. Separating these two concepts into different instructions may prove useful in increasing the performance of parallel computers. Overlapping costly synchronization with other operations is the most important area of research in this field.

Appendix A

Partial Gang Support

Partial gangs without the *waiting for a nonrunning process* problem described in Section 2.3.2.2 can be avoided by hints to the operating system about lock usage. Figure A.1 is an example of how a test-and-test-and-set spinlock could be coded, using an i860-style `lock` instruction, that blocks interrupts from user mode for a short time.

```
LOCK:   block-interrupts
TS:     test-and-set rL, LOCK.value
        if rL = 0 goto GOTIT

TTS:    load r0, LOCK.owner
        if ThreadData[r0].swapped-out then
            inform OS of spinning on address(LOCK)
        endif
        unblock-interrupts

        block-interrupts
        lock-query rL, LOCK.value
        if rL = 0 goto TS
        goto TTS

GOTIT:  store ID, LOCK.owner
        increment private.lock-count
        unblock-interrupts

CS:     <critical section>

UNLOCK: block-interrupts
        store           0, LOCK.owner
        release-lock    0, LOCK.value
        unblock-interrupts
```

Figure A.1: Advising the operating system about lock usage.

The user accessible block-interrupts and unblock-interrupts instructions allow the test-and-set and the setting of `LOCK.owner` and `private.lock-count` to be performed in one atomic operation with respect to the local processor. These are not atomic with respect to other processors, but the ordering of the synchronization operations on `LOCK.value` and the load/stores to `LOCK.owner` prevents harmful inconsistencies, as long as weak ordering is maintained.

There are many races in this code. For example, the owner of the lock may have been swapped out at the time of the residency test, but may have been swapped back in, started running, released the old lock, and acquired a new lock, in the time before the spinner has made the system call "inform OS of lock spinning." In other words, the wrong dependency may have been given to the OS. This does not really matter, because this is only an advisory system call. There is the chance that it could introduce deadlock, if the OS naively implements the hints, or dramatically slow down performance.

The OS can remove false dependencies by verifying that the lock whose address is passed to the OS by the "inform OS of lock spinning" call is owned by another processor.

The OS can use the `private.lock-count` and `private.list-of-locks-held` variables to avoid preempting processes that hold locks. These locations are written by the user without system call overhead. OS intervention is necessary only when a process has detected that the lock it desires is held by a preempted process.

Bibliography

[1] Advanced Micro Devices, Inc., *Am29000 Streamlined Instruction Processor, Advance Information*, February 1987. Publication Number 09075, Rev. A, Amendment /0.

[2] A. Agarwal and M. Cherian, "Adaptive backoff synchronization techniques," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 396–406, June 1989.

[3] T. E. Anderson, "The performance implications of spin-waiting alternatives for shared-memory multiprocessors," in *Proceedings of the 1989 International Conference on Parallel Processing*, pp. II–170–II–174, August 8-12, 1989.

[4] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," Tech. Rep. Technical Report 89-04-03, University of Washington, August 1989.

[5] T. E. Anderson, E. D. Lazowska, and H. M. Levy, "The performance implications of thread management alternatives for shared-memory multiprocessors," in *Proceedings of ACM SIGMETRICS and PERFORMANCE '89, International Conference on Measurement and Modeling of Computer Systems*, pp. 49–60, May 23-26, 1989.

[6] M. Beeler, *Inside the TC2000 Computer*. BBN Advanced Computers, Inc., February 14, 1990. Part No. A370014G10, Document Rev: A.

[7] R. Bisiani and M. Ravishankar, "PLUS: A distributed shared-memory system," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 115–124, May 1990.

[8] P. Bitar and A. Despain, "Multiprocessor cache synchronization," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pp. 424–433, June 3-5, 1986.

[9] A. Chang, M. Mergen, S. Porter, B. Rader, and J. Roberts, "Evolution of storage facilities in the AIX system," in *IBM RISC System/6000 Technology*, IBM Corporation, pp. 138–142, 1990. Form No. SA23-2619.

[10] A. Chang and M. F. Mergen, "801 storage: Architecture and programming," *ACM Transactions on Computer Systems*, vol. 6, pp. 28–50, February 1988.

[11] P. Chow, Ed., *The MIPS-X RISC Microprocessor*. Boston: Kluwer, 1989.

[12] N. G. de Bruijn, "Additional comments on a problem in concurrent programming control," *Communications of the ACM*, vol. 10, pp. 137–138, March 1967.

[13] Denelcor, Inc., *Heterogeneous Element Processor Principles of Operation, Technical Documentation Series B*, April 1981. Part No. 9000001.

[14] Digital Equipment Corporation, *MS780 Memory System Technical Description*. EK-MS780-TD-001, 1st edition, August 1978.

[15] Digital Equipment Corporation, *VAX 6200/6300, VAXserver 6200/6300, System Technical Users's Guide*, July 1989. Order Number: EK-620AB-TM-002.

[16] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Communications of the ACM*, vol. 8, p. 569, September 1965.

[17] E. W. Dijkstra, "Co-operating sequential processes," in *Programming Languages*, F. Genuys, Ed., London: Academic Press, pp. 43–112, 1968.

[18] J. Doughty, "Re: atomic instructions on R2000." Message-ID: <769@odin.SGI.COM>, Email: jeffd@norge.sgi.com, October 3, 1989.

[19] M. Dubois and F. A. Briggs, "Multiprocessor systems," in *ICPP88 Tutorials, Tutorial #5*, pp. 1–229, August 1988.

[20] M. Dubois, C. Scheurich, and F. Briggs, "Memory access buffering in multiprocessors," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pp. 374–442, June 2-5, 1986.

[21] S. J. Eggers, "Simulation analysis of data sharing in shared memory multiprocessors," Ph.D. dissertation, University of California at Berkeley, April 1989.

[22] M. A. Eisenberg and M. R. McGuire, "Further comments on Dijkstra's concurrent programming control problem," *Communications of the ACM*, vol. 15, p. 999, November 1972.

[23] Encore Computer Corporation, *Multimax Technical Summary*, January 1989. 726-01757 Rev. E.

[24] A. Forin, "Mips, Mach, test-and-set." Message-ID: <6565@pt.cs.cmu.edu>, October 17, 1989.

[25] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennesy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 15–26, May 1990.

[26] G. Gibson, "Draft SPURbus specification," in *Proceedings of CS292i: Implementation of VLSI Systems (Spring 1985)*, R. H. Katz, Ed., University of California at Berkeley, September 1985. Report No. UCB/CSD 86/259.

[27] A. Glew, "Test and set in memory — rest in peace," *EE497 High Performance Systems Seminar Presentations*, October 1987.

[28] A. Glew and W.-M. Hwu, "Snoopy cache test-and-test-and-set without excessive bus contention," *Computer Architecture News*, vol. 18, pp. 25–32, June 1990.

[29] A. F. Glew, "Synchronization primitive implementation including the bus abandonment lock," Master's thesis, University of Illinois at Urbana-Champaign, January 1991.

[30] J. R. Goodman, M. K. Vernon, and P. J. Woest, "Efficient synchronization primitives for large-scale cache-coherent multiprocessors," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 64–72, April 3-6, 1989.

[31] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer — Designing an MIMD shared memory parallel computer," in *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pp. 27–42, 1982.

[32] Gould, Inc., *NP1 Central Processing Unit (CPU) Model 4020 Reference Manual*, September 1987.

[33] G. Graunke and S. Thakkar, "Synchronization algorithms for shared-memory multiprocessors," *IEEE Computer*, vol. 23, pp. 60–69, June 1990.

[34] R. D. Groves and R. Oehler, "RISC system/6000 processor architecture," in *IBM RT Personal Computer Technology*, IBM Corporation, pp. 16–23, 1986. Form No. SA23-2619.

[35] R. Gupta, "The fuzzy barrier: A mechanism for high speed synchronization of processors," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 54–63, April 3-6, 1989.

[36] M. Herlihy, "A methodology for implementing highly concurrent data structures," in *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, March 14-16, 1990.

[37] P. Hester, R. O. Simpson, and A. Chang, "The IBM RT PC ROMP and memory management unit architecture," in *IBM RT Personal Computer Technology*, IBM Corporation, pp. 48–56, 1986. Form No. SA23-1057.

[38] C. Hunter, *Series 32000 Programmer's Reference Manual.* Englewood Cliffs, New Jersey: Prentice-Hall, 1989.

[39] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing.* New York: McGraw-Hill, 1984.

[40] The Institute of Electrical and Electronics Engineers, Inc., *An American National Standard — IEEE Standard for a Simple 32-Bit Backplane Bus: NuBus*, 1987.

[41] The Institute of Electrical and Electronics Engineers, Inc., *An American National Standard — IEEE Standard for a Versatile Backplane Bus: VMEbus*, 1987.

[42] The Institute of Electrical and Electronics Engineers, Inc., *FUTUREBUS+ P896.1 Logical Layer Specifications*, July 16, 1990. Draft 8.3.

[43] The Institute of Electrical and Electronics Engineers, Inc., *SCI, Scalable Coherent Interface, Cache-Coherence-Overview, Draft 0.75, P1596/Part III-A*, November 5, 1990.

[44] The Institute of Electrical and Electronics Engineers, Inc., *SCI, Scalable Coherent Interface, Logical Specification, P1596: Part II / D0.77*, November 10, 1990.

[45] Intel Corp., *i486 Microprocessor Technical Summary*, April 1989. Order number 240440-001.

[46] Intel Corp., *i860(TM) 64-Bit Microprocessor*, April 1989. Order Number 240296-002.

[47] G. Kane, *MIPS R2000 RISC Architecture.* Englewood Cliffs, New Jersey: Prentice-Hall, 1987.

[48] D. E. Knuth, "Additional comments on a problem in concurrent programming control," *Communications of the ACM*, vol. 9, pp. 321–322, May 1966.

[49] L. Kohn, *Architecture of the Intel i860(TM) 64-Bit Microprocessor*, 1989. Intel Order Number 240520, presented at Compcon Spring March 1989.

[50] L. Lamport, "A new solution of Dijkstra's concurrent programming problem," *Communications of the ACM*, vol. 17, pp. 453–455, August 1974.

[51] J. Lee and U. Ramachandran, "Synchronization with multiprocessor caches," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 27–37, May 1990.

[52] T. E. Leonard, Ed., *VAX Architecture Reference Manual.* Digital Equipment Corporation, 1987.

[53] T. Lovett and S. Thakkar, "The Symmetry multiprocessor solution," August 8-12, 1989.

[54] Motorola, Inc., *Technical Summary, MC68030, Second Generation 32-Bit Enhanced Microprocessor*, 1986. BR508/D.

[55] Motorola, Inc., *VSB Specification Manual, Revision C (The Parallel Sub System Bus of the IEC 821 Bus)*, November 1986. Document code MVMESB/D2.

[56] Motorola, Inc., *MC88100 RISC Microprocessor User's Manual*, 1988.

[57] Motorola, Inc., *MC88200 Cache/Memory Management Unit User's Manual*, 1988.

[58] G. L. Peterson, "Myths about the mutual exclusion problem," *Information Processing Letters*, vol. 12, pp. 115–116, June 13, 1981.

[59] M. Raynal, *Algorithms for Mutual Exclusion*. Cambridge, Massachusetts: The MIT Press, 1986.

[60] L. Rudolph and Z. Segall, "Dynamic decentralized cache schemes for MIMD parallel processors," in *Proceedings of the 11th Annual Symposium on Computer Architecture*, pp. 340–347, June 5-7, 1984.

[61] D. Schanin, "Microprocessor cache coherency," *VLSI Systems Design*, pp. 40–42, August 1987.

[62] Sequent Computer Systems, *Balance(TM) 8000 Guide to Parallel Programming*, July 31, 1985. Man-1000-00, 1003-40425 Rev. A.

[63] O. Serlin, "Alliant Computer: What next?," *The Serlin Report on Parallel Processing*, October 5, 1987.

[64] Sun Microsystems, Inc., *SunOS Reference Manual.* Part Number: 800-3827-10, Revision A of 27 March, 1990, Sun Release 4.1.

[65] Sun Microsystems, Inc., *The SPARC Architecture Manual*, August 8, 1987. Part No: 800-1399-07, Revision 50.

[66] P. Sweazy and A. J. Smith, "A class of compatible cache consistency protocols and their support by the IEEE Futurebus," in *Proceedings of the 13th Annual Symposium on Computer Architecture*, pp. 414–423, June 3-5, 1986.

[67] G. S. Taylor, "SPUR instruction set architecture," in *Proceedings of CS292i: Implementation of VLSI Systems (Spring 1985)*, R. H. Katz, Ed., University of California at Berkeley, September 1985. Report No. UCB/CSD 86/259.

[68] S. Thakkar, P. Gifford, and G. Fielland, "The Balance multiprocessor system," *IEEE Micro*, pp. 57–69, February 1988.

[69] D. J. Vianney, J. H. Thomas, and V. Rabaza, "The Gould NP1 system interconnection," in *Proceedings of the 1989 International Conference on Supercomputing*, ACM Press, July 4-8, 1989.

[70] T.-K. Woo, "A note on Lamport's mutual exclusion algorithm," *Operating Systems Review*, vol. 24, pp. 78–80, October 1990.

[71] C.-Q. Zhu and P.-C. Yew, "A synchronization scheme and its applications for large multiprocessor systems," in *Proceedings of the 1984 International Conference on Distributed Computing Systems*, pp. 486–491, 1984.