Access Ordering and Coherence in Shared Memory Multiprocessors

Christoph Ernst Scheurich

CENG 89-19

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213 740-4475)
May 1989

Access Ordering and Coherence in Shared Memory Multiprocessors

by

Christoph Ernst Scheurich

Technical Report No. CENG 89-19

A Dissertation Presented to the
Faculty of the Graduate School
University of Southern California
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Engineering)

May 1989

This work has been supported by the National Science foundation under grants DCCR-8709997 and DMC-8505328

Abstract

Shared memory forms a convenient communication medium in a multitasking multiprocessor system. However, different multiprocessors can execute the same program in different manners, possibly yielding incorrect results because the machines adhere to different rules. Differences in behavior are due to the varying approaches of designers to attack the shared memory access latency problem in multiprocessors. In particular, the manner in which multiple copies of data are controlled and the manner in which memory accesses are sequenced, propagated, and buffered has impact on the behavior of the multiprocessor.

Three shared memory execution models, referred to as *concurrency models*, are defined. The precise properties of processors, memories, and interconnection networks are derived to comply to each of the concurrency models. The usefulness of these concurrency models is demonstrated by showing the simplicity with which their rules can be applied to allow buffering of memory accesses, implement combining networks, prove cache coherence protocols correct, and design *lockup-free* caches. Specific examples are provided, both, of a cache-based multiprocessor potentially without bottlenecks and of a cache-based multiprocessor employing *lockup-free* caches which can continue to service the processor while concurrently servicing one or several access misses. The paradigms and associated conditions presented in this thesis form a set of powerful tools allowing multiprocessor designers to concentrate on functionality while being burdened less with side-effect analysis.

1.1 Problem Statement

An actual processing element of a multiprocessor is not, if at all, much different than the processing element used in a uniprocessor. A multiprocessor's instruction set may include some additional instructions to simplify inter–processor communication and synchronization, but these additional instructions are not necessary for functionality. The real difference between multiprocessors and uniprocessors is the manner in which they are programmed and the manner in which they access shared memory to communicate.

Parallel programs, whether designed to be parallel by the programmer or parallelized by a sophisticated compiler using a sequential program as its input, are fundamentally different from sequential programs. While a sequential program executed on a sequential machine defines the exact order in which instructions are performed and memory accesses are made, this is not the case for a parallel program. In a parallel program the execution sequence of each single instruction stream (*thread*) is fixed by the program but the order in which instructions are executed globally, relative to the other threads, is usually not uniquely defined by the program itself. Only a partial ordering is enforced by the program's synchronizing elements. While the exact execution order would be repeated every time the same program is executed under the exact same conditions, factors such as real time events, operating system activities, multiprogramming, and even the exact angular position of a disk platter relative to the read/write head at execution time preclude this from happening. This property of multiprocessors makes them very difficult to work with. Running a program once may yield the correct result while running it a second time, using the same input, can yield an erroneous result. This can be due to a programming error which only has an effect under critical synchronization circumstances, or due to a subtle error in the hardware which is "parallel–context–sensitive."

The programmer or compiler intends that certain instructions within one thread are executed before some other instructions in another thread. Syn-

chronization elements (algorithms or primitives) are used to enforce some partial ordering possibilities without specifying *a single order* as the only correct one [22]. However, synchronization elements themselves are part of the program to be synchronized. Synchronization works by communication via the generation and observation of global events. In a shared memory multiprocessor these events are memory read and memory write operations.

The processor–to–memory interconnection network and the memory system itself (including caches) form the "medium" in which global events take place and through which they are propagated and observed. The necessary logical properties of this medium and the manner in which processors must interact with it to fulfill the programmer's (compiler's) expectations of the behavior of the medium are defined in this thesis.

To be able to meet certain conditions in a system, the expected behavior of the system must be well defined. In Chapter two, the common expectations of a programmer or compiler regarding a parallel system's behavior are defined. The impact of these expectations on the architectural features of a machine with respect to memories, caches, interconnection networks, and temporary buffers is explained. The rules to make a parallel processor conform to three parallelism models are given and it is shown how they can be applied. Chapter three defines these concepts for the particular case of cache–based multiprocessors assuming the *sequential consistency* model of behavior. The rules and definitions of Chapter three help an engineer design new cache coherence protocols and prove correct existing ones. Chapter four utilizes the concepts of Chapter two to define the logical requirements of *lockup–free* caches in multiprocessors. The remainder of this introductory chapter presents the basic concepts of shared memory multiprocessors and reviews related work.

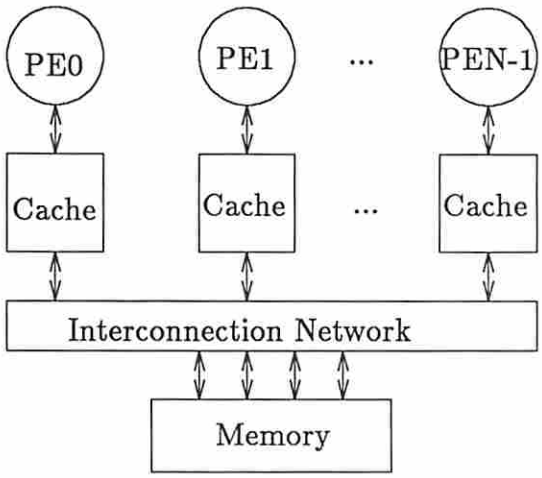


Figure 1.1: *Shared memory multiprocessor with private caches and generic interconnection network.*

1.2 Shared Memory Multiprocessors

In this thesis we limit our discussion to shared memory multiprocessors (see Figure 1.1). We define a shared memory multiprocessor as a computing system with two or more processors all of which are capable—without software aid—of accessing and modifying some shared memory locations. Processors can also have access to private memory and the system may implement direct processor–to–processor communication, but for the remainder of this thesis we assume, unless specifically stated not to be the case, that processor–to–processor communication takes place via shared memory locations.

1.2.1 Programmability

Shared memory is assumed because it represents a simple extension of the sequential von Neumann architecture to multiprocessors. Furthermore, it is easier to implement other parallel computing paradigms when starting with a shared memory architecture than vice versa. For example, a distributed system

can be emulated easily by a shared memory multiprocessor by partitioning the shared address space and by using inter–processor interrupts and shared memory mailboxes. While programming any type of MIMD (Multiple Instructions Multiple Data) parallel computer is never simple, shared memory multiprocessors present the programmer and compiler with an easy to understand architecture and model of computation which does not necessitate the learning of new inter–process(or) communication primitives, while not excluding their implementation if wanted. Some discourse on the relative merits of shared memory versus distributed computers can be found in [33], as can be found introductory materials on most issues involving parallel systems.

1.2.2 Memory coherence

The presence of shared memory in multiprocessors does not dictate that all processors must access the same physical device to read or modify a particular memory location. Using a single physical device to store a given shared memory word would result in performance degradations due to contention and overall memory access latency—*rendering the design inefficient.* Hence, either each processor maintains its own physical copy of all shared data (replicated memory), or maintains copies of select data items (caches), or multiple accesses to the same shared location are merged before reaching main memory (combining networks). While private and shared caches are common, fully replicated memories are not presently used, and recombining networks are becoming more popular (they are discussed separately in Section 1.4.4).

In the cases of private caches and replicated memories the issue of *memory coherence* must be addressed. Memory coherence was defined by Censier and Feautrier as a property of a shared memory system as follows [12].

- A memory scheme is coherent if the value returned on a load instruction is always the value given by the latest store instruction with the same address.

This definition relies on the ability to precisely order read and write accesses to the same shared writable word. In multiprocessors where several copies of a word can exist in different caches it is not always possible to define such an order. For example, is a load performed once the processor decodes the instruction and determines the address, or is it performed when the access has missed at the local cache and the datum must be retrieved from memory, or is the load only performed once the load request and value of the word have been irreversibly bound? Likewise, is a store performed once address and value have been bound, or only after the local cache has been updated, or even later once every cache has been updated or invalidated? Censier and Feautrier's definition is only easily applicable if an absolute order of accesses is apparent. This is the case, for example, if a central controller (central directory, in the case of a cache–based system) serializes all accesses to a given word which must be maintained coherent. If no centralized "serializing" device for accesses exists, then the above definition of coherence is very difficult to apply.

1.2.3 Virtual memory

Most modern computing systems, whether sequential or parallel, utilize virtual memory. In the context of this thesis virtual memory is not an important issue. The implementation of virtual memory adds an extra layer of indirection to the address–data matching process but this process is well understood and poses only implementational problems. Unless otherwise noted, we assume in this thesis that all addresses are physical addresses and usually do not discuss the implications of page faults. This assumption is not unrealistic if virtual–to–physical address translation is performed in the immediate vicinity of each processor. This is usually accomplished by a small cache called the *translation lookaside buffer* (TLB), which buffers recent virtual–to–physical address translations.

In cache–based systems, the existence of a TLB before or in parallel with the cache indicates that the cache is addressed using physical addresses—

this is the norm. Lately, efforts have been expended to use virtual addressed caches. By doing this, the extra translation before a cache access is avoided and hence the cache access latency is reduced. Using virtual addressed caches, however, poses the *synonym problem.* Cache synonyms are defined as two or more different virtual cache entries which map to the same physical word or block. The problem of synonyms is orthogonal to the topic of this thesis and will not be further discussed.

1.2.4 Examples of existing and proposed multiprocessors

Commercial and experimental shared memory multiprocessors have been built or are presently being designed. Early commercial products include the IBM 370/168MP [52], the IBM 3090 system [64], and the Denelcor HEP [60]. Early experimental systems were the Cm* and the C.mmp [26,27]. Newer systems include the NYU Ultracomputer [31], the CEDAR project [28], and IBM's RP3 [50]—all three are experimental computers—and new commercial systems are, for example, the Sequent Balance, and Cray's YMP model.

1.3 Cache Memories

Cache memories are commonly used in shared memory multiprocessors. The purpose of caches is to buffer frequently used data close to the processor, resulting in both faster access times and less main memory contention. Caches come in different sizes, can be partitioned and addressed differently, and may use different algorithms to replace present data to make space for new data. If caches are private and can contain shared, writable data then the problem of maintaining cache coherence must also be addressed.

From the context of shared memory multiprocessor design, many of the organizational details of caches are unimportant. Smith provides a good tutorial on these specifics and their impact on performance [59]. A more recent paper by

Smith discusses at length the impact of different block sizes on cache performance [61].

1.3.1 Cache coherence

Only private caches are of interest here. If the compiler can tag all shared, writable data and the cache can recognize such tags and not cache such data, then data inconsistencies cannot occur among different caches. The efficiency of this approach depends on the overall proportion of shared writable data accesses by all processors. Lee *et al.* cite percentages of noncachable data for various workloads in the range of 0–63% [45]. In this case the tagging was performed by the PARAFRASE compiler [67]. The usefulness of a cache depends on the cache hit rate it can attain. The percentage of noncachable accesses represents the lower bound on miss rates for systems using data tagging (if the hit rate is defined as the proportion of accesses fulfilled at the cache versus the *total* number of accesses made). Hit rates of less that 85% are highly undesirable and can lead to grave performance degradations.

Ever since the first implementation of a hardware–based cache coherence mechanism, researchers have worked on finessing protocols to generate the least cache–external traffic.

- One of the first cache coherence schemes was implemented in the IBM 370/168 MP (introduced in 1974). The two processor, two cache system used dedicated processor–to–processor lines to communicate invalidations on *every* write operation [52].

- To reduce the amount of cache–external memory traffic and to reduce cache access latencies, Tang proposed a central directory scheme [63]. A block in cache could be in either state *shared* or in state *private*. The central directory maintains duplicates of all cache directories and controls transitions. Using this scheme, write operations on private blocks do not cause cache–external traffic.

- Censier and Feautrier proposed a coherence scheme very similar to Tang's while reducing the amount of state information which must be maintained per block by the main memory or central directory [12]. In this case the central directory tracks the status of *every* memory block rather than only cached blocks. The resulting protocol has a more distributed nature.

- The popularity of distributed cache coherence schemes increased, and in 1983 Goodman proposed an new version of the distributed protocol which does not invalidate all copies of a block upon the first modification of the block [30]. The protocol is now known as the *write–once* protocol. The protocol was designed with realistic block access patterns in mind.

- In 1984 new protocols were introduced by Archibald and Baer [4], by Papamarcos and Patel [48], and by Rudolph and Segall [51]. The latter protocol introduced the concept of bus interruption. A snooping cache, in this protocol, can interrupt a request present on the bus to force a memory write–back before the access can be retried.

- The description of a commercial cache–based multiprocessor using a distributed coherence scheme was also published in 1984 by Frank [25].

- Katz *et al.* published a distributed protocol in 1985 which improved upon Goodman's 1983 scheme [34]. The protocol is now referred to as the *Berkeley* protocol.

- To further reduce cache–external traffic, some protocols attempt to anticipate typical access patterns and use selective write–broadcasts. A popular example is the Dragon protocol [47].

1.3.2 Multiprocessor cache performance

Correctness and efficiency are not disjoint in multiprocessor cache designs. It is easy to guarantee the "correct" operation of a multiprocessor at the

cost of reduced performance. Knowledge about the proportion of shared writable data accesses and the distribution of such accesses in characteristic workloads can provide important clues to the designer of shared memory systems. Performance analyses allow designers to make only prudent tradeoffs.

Much work has been done on performance evaluations of uniprocessor caches. Such performance analyses must be based on typical processor workload assumptions. Workloads can be obtained by tracing addresses and access types (read or write) for what are assumed to be typical programs. Different tracing methodologies exist [1]. Traces can be captured by hardware monitors, operating system facilities, or by using machine simulators to execute a program to be traced. Alternatively artificial workloads can be designed using compilations of typical program behavior statistics [62].

Multiprocessor address traces are much more difficult to obtain because few benchmark parallel programs exist. Even if multiprocessor traces are available the order in which different processors interleave their accesses is not fixed. This inter–processors access order is subject to both timing which is dependent on the architecture and synchronization barriers. Consequently, performance analyses of multiprocessors are usually based on broad assumptions, yielding results that must be interpreted in the context of the assumptions.

An analysis on the effect of maintaining coherence has been done by Dubois and Briggs [18]. The study was based on an artificial workload model and Markov analysis. Archibald and Baer performed studies on common bus–based cache coherence protocols using a multiprocessor simulator [5]. The studies verified that refinements in protocols do have some impact on the overall performance of multiprocessors. Using infinite cache models, Dubois and Wang have studied the effects of invalidations on system performance [20,21]. The results yield the upper bounds on hit rates for invalidation–based protocols for the assumed workloads. Yang and Bhuyan developed a queueing network model for a modified *write–once* protocol operating on a multi–bus, packet–switched interconnection [68]. Some performance figures of the bus–based Sequent Symmetry

multiprocessor were published by Lovett and Thakkar [46]. The results verified the apparent advantage of write–back caches over write–through caches. Using trace-driven simulations, Eggers and Katz [23] compared the performance of several cache coherence protocols for different parallel applications. The work focuses on the *sharing behavior* of the different applications, concluding that sharing is minimal and derives a model based on the behavior of the examined applications. Agarwal *et al.*, revisit central directory schemes and compare their performance to snoopy cache protocols in [2].

1.4 Interconnections

The interconnection network allows processor and memories to communicate. The issue of interconnection is *very* important in the context of this thesis. While the usual concerns regarding interconnection networks are speed, bandwidth, cost, and reliability, we are concerned with the logical properties of different interconnection types. Since the interconnection network is the medium through which events propagate and are observed, it is important to understand how the interconnection can affect the relative order of these events.

1.4.1 Bus–based interconnections

Single circuit–switched buses form the simplest type of interconnection. Circuit–switched buses constitute a broadcast medium which guarantees absolute serialization of all accesses. The order in which a single bus accepts accesses is also the order in which the accesses reach all destinations. The advantage of this property is that devices connected to the bus can "snoop" and observe all global events in the same order.

Packet–switched buses do not necessarily guarantee that all connected devices observe all bus events in the same order. Since multiple transactions can take place concurrently on a packet–switched bus devices can observe events caused by different sources in different orders. However, the events triggered by

a single device are usually always observed in the same relative order by all other devices.

The behavior of multiple buses depends on the manner in which a source accesses them. For example, if each bus at each device has its own access queue, then accesses by a single source may be serviced in a different order than the order in which the accesses were placed in the queues. If the buses are circuit–switched, then all devices will still observe all accesses in the same order. If packet–switched buses are used, this may not be the case.

1.4.2 Other interconnection networks

Many other types of interconnection networks exist. For high throughput such networks often are packet–switched and incorporate buffers to be able to flexibly adjust to differences in loads. To avoid bottlenecks and improve reliability such networks often provide multiple paths between source and destination pairs. The particular path chosen for a communication may be determined by timing, randomly, or based on load characteristics. It may also be possible for communications to fail completely if switch buffers are full. Such an event would cause an error message to be sent to the sender of a request.

In this thesis we call an interconnection *generic* if it has all of the following properties:

- packet–switched,
- contains buffered nodes,
- multiple source–to–destination paths exist,
- and the path a request will take is unknown at the time of sending it.

Such an interconnection network can be viewed as a space which propagates messages with a random (but larger than some minimum value) delay. Examining this type of network is interesting because it represents the most general case of an interconnection.

1.4.3 Combining networks

Combining networks are used to decrease network load and access latencies of memory requests [31]. Multiple memory read operations of the same memory location can collide at some node before memory and are combined into a single request to memory; upon return of the word the value is distributed back to all requesting processors. The effect on the logical (generic) network property is that the access delay from the combining node to memory and back appears to be the same for all accesses.

When a read and a write operation collide at a node, the read need not propagate to memory but is assigned the value defined by the write and this value is returned to the reading processor. The write has to continue to the destination memory. The logical effect of this sequence of events is that the read combining with the write may return an *early* copy of the word, one not yet available to the remainder of the processors.

When a write collides with another write request to the same word, one of the two is eliminated. The logical effect is that the write operation which proceeds to main memory logically occurred after the eliminated write operation.

1.5 Synchronization and Event Ordering

The processors of a multiprocessor must synchronize to execute programs correctly. While different types of synchronization exist, such as mutual exclusion and barrier synchronization, the essence of synchronization is always that a processor is not permitted to proceed until some (externally triggered) condition has been met. The condition is caused by either one or several other processors. In a shared memory multiprocessor, where all global states are represented by shared memory states, the condition to be met is indicated by one or several shared memory word states.

1.5.1 Synchronization

Three types of program synchronization are possible. We call these synchronization types *explicit* synchronization, *implicit* synchronization, and synchronization *by mutual exclusion.*

1. Explicit synchronization relies on a complete matching of every read operation of a shared writable memory location with a corresponding write operation of this memory location. That is, a read cannot proceed until the specific write associated with this instantiation of the read has been performed (whether by the same or a different processor). Likewise a write operation to a given memory location may not proceed until the specific number of read operations accessing the memory location before the write have been performed. Such a machine does not need a program counter and is based on the *data–flow* principle of operation.

2. Implicit synchronization relies on the fact that the sequential execution of each parallel thread can enforce an implicit ordering among the threads. Many well known synchronization protocols, such as the Bakery algorithm [17,39], function on the basis of implicit synchronization. As we will show later, implicit synchronization relies on some machine characteristics.

3. Synchronization by mutual exclusion is based on inter–process serialization. Once within a critical section a process(or) can read and write data without further synchronization because it is the only processor with access to any shared writable data addressed within the critical section. Acquisition and release of critical sections is accomplished by using special synchronization accesses for which ordering is enforced more strictly than is necessary for regular memory accesses.

1.5.2 Event ordering

It is important to realize that in a multiprocessor real time order has little importance. An event does not occur at the time that it takes place at its "source" but rather takes place relative to when the event becomes observable at a destination. That is, an event which cannot have any bearing on a processor—it has not yet been observed—has not yet actually occurred relative to that processor. A cache-based, shared memory multiprocessor with a generic interconnection, for example, may not conform to a sequential model of execution (one which humans tend to assume). Since events take place relative to when they become observable, and since observation is dependent on the communication delay of the network (which is random in a generic network), events can be observed in different orders at different processors[1]

Collier has studied the differences of behavior of multiprocessors caused by differences in the propagation order of accesses [13]. He defines two multiprocessor architectures as *distinguishable* as follows:

- Two architectures A1 and A2 are called distinguishable if there is an execution such that the graph of the execution is linearizable under A1, but not under A2. To prove two architectures are distinguishable, find such an execution (that is, a program with input and output data), such that (1) the program can compute the output data from the input data under architecture A1, but cannot under architecture A2 [14].

A *linearizable* execution is one which can be reproduced on a uniprocessor using a single copy of main memory. Cyclic relationships cannot be deduced from such

[1] The notion of relative differences in the order of observed events is nothing new to physicists. In the study of relativity two events occurring simultaneously within one frame of reference are not simultaneous if occurring in different frames of reference—one moving with respect to the other [24]. The skew in simultaneity affects the relative order of multiple events. While the processors within a multiprocessor do not move, the inter-processor (or processor-to-memory) communication delay is random which is equivalent to fixing the message propagation speed and varying the memory-processor and processor-processor distances continuously and randomly—which is of course equivalent to "movement".

an execution. (A cyclic relationship is for example: a must have occurred before b, b must have occurred before c, but c must have occurred before a.)

The notion of distinguishable architectures is important because a simple multiprocessor can more easily be shown to conform to an expected set of behavior rules than a complex one. If the complex machine can be shown to be indistinguishable from the simple one then it will also conform to the expected behavior rules of the simpler machine. Collier defines a set of rules, *order rules, atomicity rules*, and *multigraph rules.* These rules can be used to test whether two architectures are distinguishable. The document [14] is in preparation and the interested reader should consult it when published.

While Collier's work can be used to draw some important conclusions about multiprocessor behavior it is difficult to apply the work (as we are aware of it now) to practical systems. The exact effects of cache coherence protocols, non–FIFO buffering, and recombining networks are difficult to integrate into these concepts. It is also not simple to use these concepts to derive new and more efficient architectures.

Other work on ordering of events has been done by Lamport. In [42], Lamport defines *sequential consistency* as a property of an execution. A sequentially consistent execution on a multiprocessor is one in which all memory accesses within a single execution thread are performed in program order and in which memory is coherent. More generally, Lamport defines sequential consistency as following:

- The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Sequential consistency is an important concept yielding a powerful model of parallel system behavior. In some cases proving a system sequentially consistent is difficult, and in other cases sequential consistency may not be efficient.

In the context of distributed systems, Lamport has also developed a scheme to dynamically determine order without the use of real time clocks [41]. Logical clocks can be used to determine precedence relations in a completely distributed manner. While the concept of logical clocks is more applicable to distributed computing systems, it can be used in shared memory multiprocessors to re-order the observation of events at a destination. For example, an observed event can be ignored by a processor until another event which should have occurred before has been observed as well. In [43,44], Lamport further explores the intricacies of interprocess communication, but does not apply his results to the problems created by shared memory systems.

The concept of sequential consistency is used by Shasha and Snir to define necessary delay parameters for a given architecture to make the architecture correct [57]. In this work, a rigorous method is described to determine the minimal delay between two shared memory accesses within a single execution thread to ensure sequential consistency. The developed method can be used by a compiler to include delay statements in object code intended for a given architecture. However, it is assumed that only single copies of data exist.

Some parts of this thesis have been presented before. A very early version of Chapter two appears in [19]. The effects of event orderings were discussed in [22]. Chapter three was first published in [53]. Chapter four is the synthesis of two papers [54,55]. For those readers interested in the chronological thought-processes resulting in this thesis, the above cited articles may be of interest.

Chapter 2

SHARED MEMORY IN MULTIPROCESSORS

The user of a computer has certain expectations about the manner in which it executes programs. These expectations are guided by the system's *logical model of behavior.* In the case of a shared memory multiprocessor we refer to the logical model of behavior which specifies event-time-location ordering as the *concurrency model.* Some parallel programs which execute correctly on a machine adhering to one concurrency model will not, in general, do so on another machine, compliant with a different concurrency model. Nonetheless, both systems can be useful. The concurrency model of a multiprocessor depends on the behavior of the underlying architecture.

In a shared memory multiprocessor, the exact behaviors of two entities have impact on the concurrency model; namely the behavior of individual processors and the behavior of the shared memory system. We call these *behaviors* the *processor model* and the *memory model*, respectively. It is important to match the processor model and the memory model to obtain a useful concurrency model which is easily programmed and can be unambiguously defined for the user.

In this Chapter we investigate different processor and memory models, define resulting concurrency models, and explain what type of programs will

execute correctly under a concurrency model. To ease programming, which is a complex task in any parallel environment, concurrency models must be relatively simple. Many high–performance mechanisms, commonly used in uniprocessors (caches and pipelining, for example) and likely to be used in multiprocessors, make the actual behavior of such systems far from simple. The key to the successful integration of these mechanisms into actual architectures is to make them "invisible" to the user of the machine. How to do this is defined later in this Chapter.

2.1 Event Ordering in Multiprocessors

Computers execute instructions listed in the program. The outcome of an instruction depends (usually) on the present "state" of the machine. The outcome itself is (usually) a modification of the previous state. If the execution of instructions would not alter the system's state, then instructions could be executed in any order. This is not the case in shared memory multiprocessors.

In the context of this thesis, the state of shared memory is of interest. Shared memory is accessible and modifiable by all processors. The order in which accesses to shared memory are made, and the order in which they take effect affects the outcome of executions. The focus is on shared memory because it represents the relevant system state which can be affected by more than one processor. Other system states that can affect and be affected by more than one processor are ignored. In an actual system implementation these factors, such as inter–processor interrupts and dynamic file sharing, have to be taken into consideration as well; but this does not diminish the relevance of shared memory functionality.

The notion of "order" is well defined in a sequential environment such as a uniprocessor. In a complex multiprocessor, executing a parallel program, this is not the case. Nonetheless, processors in a multiprocessor must affect each

other in some *order* to synchronize and communicate. A processor is affected externally by an *event*.

2.1.1 Events in multiprocessors

We define an *event* in a multiprocessor as an action that can have an effect. Both memory write and memory read accesses constitute events. A write is an event because at the instant it has taken place it can begin to affect processors. A read is an event because the value returned to the processor is likely to affect it and its future activity.

In the context of this thesis, only shared memory interactions are of interest. Hence, we consider shared memory and the interconnection to shared memory as the "medium" through which events propagate and in which events take place. This means that all other external or internal activities which can affect a processor but which do not use shared memory as their medium will be ignored. Most of these other activities are private to a single processor. For example, the reading or writing of a processor register can by itself only affect the processor which the register belongs to. Other activities which are not private—they are global—can conceptually be modeled as shared memory activities. For example, the setting of an inter–processor interrupt can be viewed as the writing of a single shared memory location (by the interrupting processor) and the consequent reading of the memory location (by the interrupted processor).

The only global events which are of interest are shared memory *read* and *write* operations. More specifically, the only relevant global events are write operations to memory locations which are readable by at least one other processor besides the writing processor, and read operations from memory locations which can be modified by at least one other processor which is not the reading processor.[1]

[1] It would be more precise, but not more useful, to only define a memory write operation as an event and to define a read operation as the *actualization* of the write which defined the value the read returns. This is due to the fact that a "new" value in a memory location which is guaranteed never to be read by a processor, has never occurred for all practical purposes, and

A write by another processor only has relevance to a processor once the updated value caused by the write becomes available. At the time when this is the case the write event becomes "observable"—it can affect the processor. In the remainder of this thesis the notion of "observing a write" will appear repeatedly. This means that the updated value due to a write becomes available to a processor; it does not mean that the processor actually reads the new value—this would imply a separate read event.

2.1.2 Event ordering

In a shared memory multiprocessor memory access events are caused by instruction execution. The order in which these events can occur and be observed depends on the programs executed and on the architecture of the parallel machine.

1. The order in which instructions belonging to different instruction streams are executed is not fixed in a concurrent program. If no synchronization among instruction streams exists, then a very large number of different instruction interleavings and hence event interleavings is possible.

2. If for performance reasons the order of execution of instructions belonging to the *same* instruction stream is different from the order implied by the program, then an even larger number of event interleavings is possible.

3. If write accesses must propagate to different replicated copies of the accessed word (as could be the case in a cache–based system) and if a processor may access an updated copy of a word before *all* other processors have the updated word available, then different processors may individually *observe* different event interleavings during the same execution. In this

hence the write which defined this "new" value was not an event (using our definition of *event*). It is more practical, for our purpose, though, to assume that reads and writes are both global events.

```
-------------------------------------------------------------
        P1              P2              P3
      ----------------------------------------
      a: A:=1         c: B:=1         e: C:=1
      b: PRINT BC     d: PRINT AC     f: PRINT AB
-------------------------------------------------------------
```

Figure 2.1: *Three program segments to be executed in parallel.*

case the total number of possible event interleavings of a program becomes larger again.

To illustrate different types of interleavings which are conceivable, we examine three program segments to be executed concurrently by three processors (initially $A = B = C = 0$, and we assume that a Print statement reads both variables indivisibly during the same cycle), as shown in Figure 2.1. If the outputs of the processors are concatenated in the order P1, P2, and P3, then the output forms a six-tuple. There are sixty-four possible output combinations. For example, if processors execute instructions in program order, then the execution interleaving (a,b,c,d,e,f) is possible and would yield the output 001011. Likewise, the interleaving (a,c,e,b,d,f) is possible and would yield the output 111111. If processors are allowed to execute instructions out of program order, assuming that no data dependencies exist among reordered instructions, then the interleaving (b,d,f,e,a,c) is possible and would yield the output 000000. Note that this outcome is not possible if processors execute instructions in program order only.

Each program segment causes a write event and a read event (actually two read events take place per PRINT statement, but since we assume indivisibility of the two reads we can consider them as a single read). Of the 720 (6!) possible event interleavings, 90 preserve the individual program order. It has already been pointed out that out of the 90 program-order interleavings,

not all six-tuple combinations can result (i.e., 000000 is not possible). The question remains whether out of the 630 non-program-order interleavings all six-tuple combinations can result. So far the assumption has been made that the memory system of the example multiprocessor behaves in a manner such that every processor "observes" memory updates in the same order. This may not be the case in a cache–based system, for example.

If writes are observed in the same order by all processors then it is easy to show that, indeed, not all six-tuple combinations are possible, even if processors do not adhere to program order. For example, the outcome 011001 implies the following: Processor P1 observes that C has been updated and B has not been updated yet. This implies that P3 must have executed statement (e) before P2 has executed statement (c). Processor P2 observes that A has been updated before C has been updated. This implies that P1 must have executed statement (a) before P3 has executed statement (e). Processor P3 observes that B has been updated but A has not been updated. This implies that P2 must have executed statement (c) before P1 has executed statement (a). Hence, (e) occurred before (c), (a) occurred before (e), and (c) occurred before (a). Since this ordering is plainly impossible, we can conclude that in a system, in which writes are observed in the same order by all processors the outcome 011001 cannot occur.

The above conclusion does not hold true in a multiprocessor with replicated memories in which updates pass through a generic interconnection. Let us assume that the actual execution interleaving of instructions is (a,c,e,b,d,f). Let us further assume the following sequence of events. When P1 executes (b), P1's own copy of B has not been updated, but P1's own copy of C has been updated. Hence, P1 prints the tuple 01. When P2 executes (d), P2's own copy of A has been updated, but P2's own copy of C has not been updated. Hence, P2 prints the tuple 10. When P3 executes (f), P3's own copy of A has not been updated, but P3's own copy of B has been updated. Hence, P3 prints the tuple 01. The resulting six-tuple is indeed 011001. Note that all instructions were *executed*

in program order but other processors did not observe the resulting events in program order.

We may ask ourselves whether a multiprocessor functions incorrectly if it is capable of generating any or all of the above-mentioned six-tuple outputs. Clearly the answer to this question depends on the expected concurrency model and the programming "style" used to enforce the "correct" execution of a parallel algorithm. Before defining what is correct and what is wrong in a multiprocessor it is important to first understand the behavior of a uniprocessor.

2.2 Processor and Memory Behavior in Uniprocessors

In a uniprocessor the processor and memory behavior can be discussed together due to the simplicity of the resulting *logical model of behavior.* A uniprocessor executes instructions sequentially in program–order and accesses data memory as instructions demand in execution–order. Memory is a simple "black box" which serves the purpose of returning the value associated with the most recent write operation upon a read operation accessing the same memory location. Interrupts can affect this simple logical model of behavior, but we will ignore their effects for the time being.

The logical model of behavior and the actual physical behavior of a uniprocessor can differ substantially. This is particularly the case for sophisticated processors which are designed for high throughput. The inherent sequentiality, implied by the logical model of behavior, technological limitations, and cost factors make it practical to deviate from the strict logical model of behavior when implementing a uniprocessor. This poses no problem as long as the behavior of the machine, as can be observed by the operator/programmer via the *usual means*, does conform to the logical model of behavior. The "usual means," in this case, refers to the machine I/O devices intended to provide man–machine communication; "other means" such as hardware probes or software accessing

special diagnostic circuits in the machine will reveal the difference in physical and logical behaviors. Some common differences between the logical and the physical architecture and behavior of a uniprocessor are caused by, for example, caches and pipelined execution.

The essence of the logical behavior model of a uniprocessor can be defined by three rules:

Compute Rule: Processors compute correctly. Arithmetic and logical operations are performed by the rules expected by the programmer or compiler and as defined by the instruction set.

Sequencing Rule: Instructions are executed one–after–another, in program order. Memory accesses are issued one–by–one to the memory system in the order generated. An access is issued only once the previous one has been completed.

Memory Rule: The memory system accepts two types of accesses. A read access to memory location M returns the value written into that location by the most recent write access of memory location M. Memory accesses are serviced one–by–one in the order made by the processor.

The *compute rule* is stated for the sake of completeness and will not be discussed any further in this thesis. We assume that any processor, whether part of a uni– or a multi–processor adheres to it. Furthermore, the memory rule refers to words and we assume that a *word* is the unit of data which is used as a basic external "state unit." The only implication of this assumption is that copies of words are always updated atomically. That is, a copy of a word is not accessible until all its bits have been updated to the new value. Note that this assumption does not preclude partial–word write updates.

2.3 Processor Behavior in Multiprocessors

Clearly, the individual processors of a multiprocessor must be able to correctly execute sequential programs. Hence, the *sequencing rule* must also be upheld in multiprocessors. We could therefore state that the processor model of the processors in a multiprocessor is the same as the processor model of a uniprocessor. This statement is not incorrect. Any multiprocessor which precisely adheres to this rule, behaves in a manner which is compatible with *any* concurrency model we are aware of.

It is also the case that in uniprocessors the sequencing rule is frequently broken. Uniprocessors often overlap the execution of instructions or even execute instructions out of program order. This is done for performance reasons. Memory accesses can also be issued out of order or be made before the previous access has been completed. Again, enhanced performance is the reason for allowing this. When uniprocessors do break the sequencing rule it is done in a way that has no impact on the outcome of an execution: Dependencies within the execution stream are observed. Two types of dependencies exist. Namely data dependencies and control dependencies. Control dependencies are internal to the processor and will not be discussed further.

Data dependencies, in the context of uniprocessor memory accesses, are referred to as *hazards*. Hazards exist if the out–of–order or overlapped issuance of memory accesses alters the result of the execution. In uniprocessors three hazards can be identified.

1. Read–after–write (RAW) refers to the hazard encountered if a read follows a write to the same memory location. Reversal of the access order will yield the wrong value for the read.

2. Write–after–read (WAR) refers to the hazard encountered if a write follows a read to the same memory location. Reversal of the access order will yield the wrong value for the read.

3. Write–after–write (WAW) refers to the hazard encountered if a write follows another write to the same memory location. Reversal of the access order will yield a wrong memory state.

Uniprocessors which break the sequencing rule *appear* to adhere to it if they avoid hazards. We call the modified sequencing rule *hazard–free sequencing* and rename the original sequencing rule to *strict sequencing.*

Processors of a multiprocessor can either adhere to strict sequencing or to hazard–free sequencing. As will be shown later, the choice has impact on the possible concurrency model.

2.4 Memory Behavior in Multiprocessors

The memory rule used to define memory functionality of a uniprocessor needs to be refined to take the possibility of concurrent memory accesses by different processors into account. As was the case with the sequencing rule, the memory rule must remain the same with respect to single processor accesses. This has to be the case so that the processors of a multiprocessor can correctly execute sequential programs.

2.4.1 Strictly coherent memory

A memory rule for parallel processors with private caches was defined by Censier and Feautrier [12]. It can be applied whether or not private caches are used in the architecture. The rule is called *memory coherence* but it will be referred to as *strict coherence*:

Definition 2.1 (Strict Coherence) *A memory scheme is coherent if the value returned on a load instruction is always the value given by the latest store instruction with the same address.*

The memory coherence rule is an obvious extension of the uniprocessor memory rule—the difference being that it implies a total order on memory accesses to the

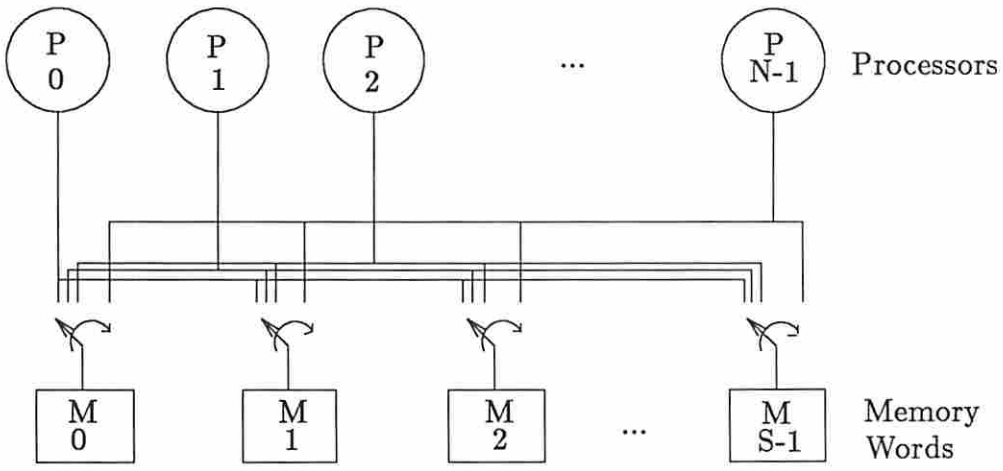


Figure 2.2: *Logical architecture of a strictly coherent multiprocessor memory system.*

same word *throughout* the system. Without the ability to define a global *order* of updates on a word, the notion of *latest store* cannot be applied.

Therefore, a multiprocessor memory system which strictly adheres to the memory coherence rule *must* have a serializing device which imparts an *order* on all accesses to the same word. Figure 2.2 demonstrates this concept. The *rotating switches* permit access to a word to one processor at a time only. When a switch is closed with respect to a processor one or a sequence of accesses may be made to the word with memory adhering to the uniprocessor memory rule. A switch may not *rotate* before the previous access has been completed.

In an actual architecture implementation, the separate serialization of every single word is infeasible. Memory coherence can also be maintained if all accesses to a *set* of memory locations are serialized (i.e., an interleaved memory system). In the extreme case, a single memory module can contain all of memory and *all* accesses to it are globally serialized. While this appears to be the obvious memory model it actually represents an extremely restricted coherent memory model. In either case the behaviors of the memory systems are indistinguishable.[2]

[2]It can be argued that this is not the case if processors use hazard–free sequencing that allows two or more accesses to be issued concurrently. In the case of multiple serializing memory

At this time we introduce a visual aid to better understand multiprocessor memory models. Each memory word of shared memory will be updated a number of times throughout the execution of a parallel program. We can graphically depict these updates as a column of "stacked boxes" in which the top box stands for the initial content of the word and the bottom box stands for the terminal content of the word. The intermediate boxes stand for different values the word contained during the execution. If n write operations were performed to a word by all processors combined during an execution, then the number of boxes is n+1 (due to the initial content of the word). The stacks of each memory word can be placed next to each other to model the total shared memory history of an execution. Figure 2.3 depicts such a total memory history.

Each time a processor executes a shared memory load operation it has access to a "crossection" of the memory history. While a load only refers to a single word, it cannot be predicted which word will be addressed during a load. Hence, a load of a processor can be indicated on the memory–history diagram by drawing a line which passes through each memory state of each word which would be returned if it was the target of the load. On the left–most edge of the diagram loads by the same processor appear in descending order. (The order of loads by the same processor is the order in which the memory is accessed—it may not be program order in the case of hazard–free sequencing.) By observing how different load–lines by the same processor and different processor cross and overlap, statements can be made about the memory model. We call such "lines" *load–state lines.* To simplify the diagrams, overlapping lines are shown as parallel lines within the same grid.

---

modules, accesses to different words can then be executed out of program order. In the case of a single memory module this is not possible since *all* accesses are serialized. As will be shown later the difference is detectable. The difference, however, is not due to different logical behaviors of the memory systems but due to the way a single–module coherent memory inherently restricts hazard–free sequencing. If processors would place multiple hazard–free accesses into an access buffer, and the single memory module selects items from the buffer at random, then the difference between single– and multiple–module memories disappears.

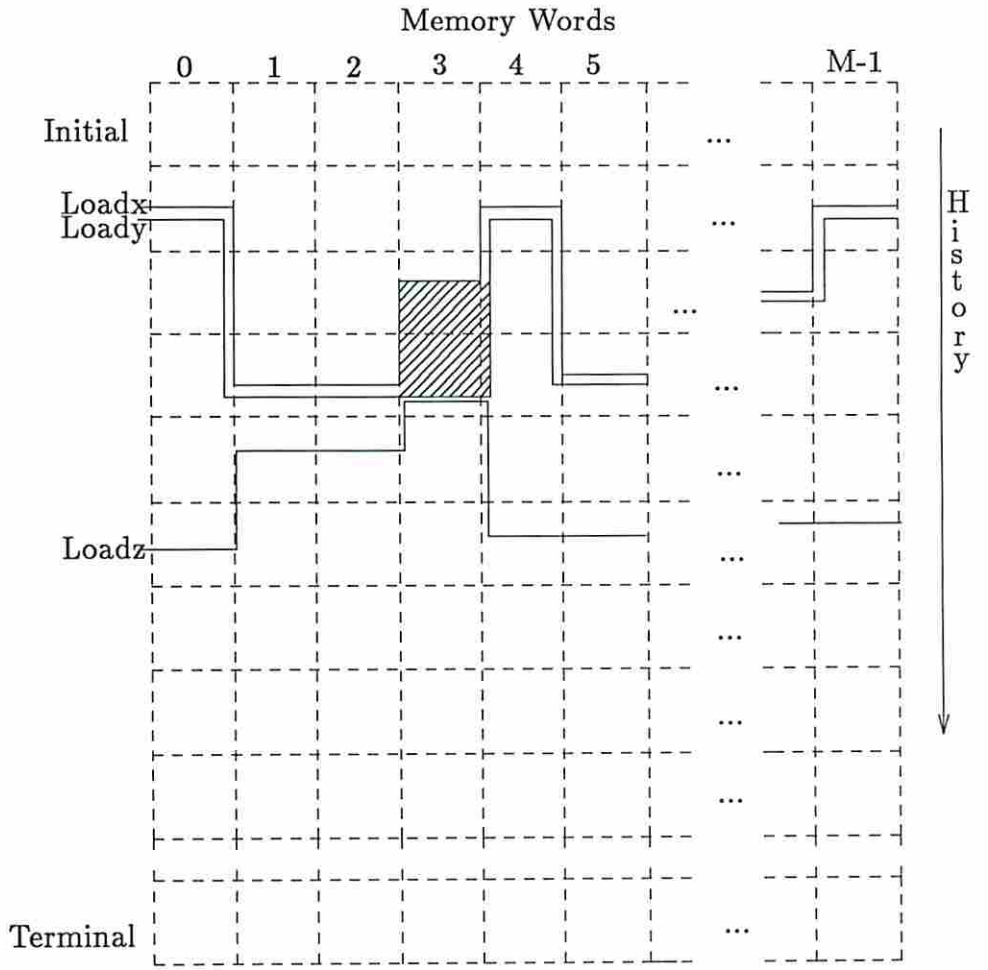


Figure 2.3: *Memory–history diagram of an M–word memory system.*

In Figure 2.3, three loads x,y, and z are shown in the memory–history diagram. Loads x and y differ only in that load y can observe a different value for memory word 3. This is indicated by the shaded area. If the diagram is correspondent to the memory–history of a system with strictly coherent memory implemented by the architecture of Figure 2.2 then each word–state stack can be defined by the actual sequence of updates of a memory cell representing a word. In a strictly coherent memory the following is true:

- Load–state lines due to the same processor can never cross.
- Load–state lines due to different processors can never cross.

2.4.2 Write ordered memory

In a multiprocessor it is desirable to maintain multiple copies of the same data in different memories. Such *cache–based* systems can be made to be strictly coherent—the definition was, after all, derived for this particular case. However, the serialization restriction on all copies of data still holds. A relaxation of this restriction will now be derived. This relaxation is not, however, targeted to cache–based systems in particular but, rather, targets the case of multiple copies of the same data in the system, in general.

Cache–based systems complicate the issues at hand for three reasons. Firstly, a cache may or may not have a copy of a particular data item present at any time. Secondly, data in caches usually have an associated state which defines the data's accessibility. Finally, data are normally "moved" among caches in units larger than words. To take these peculiarities into consideration would obfuscate the pertinent issues to be discussed. However, the following discussion *does* apply to cache–based systems once interpreted in the context of cache presence. Later in this Chapter and in Chapter three the case of cache presence is treated in full.

Take a multiprocessor such as depicted in Figure 2.4. Each processor has a fully replicated local memory which, with respect to the processor, adheres to the uniprocessor memory rule. When a processor issues a write it is performed at

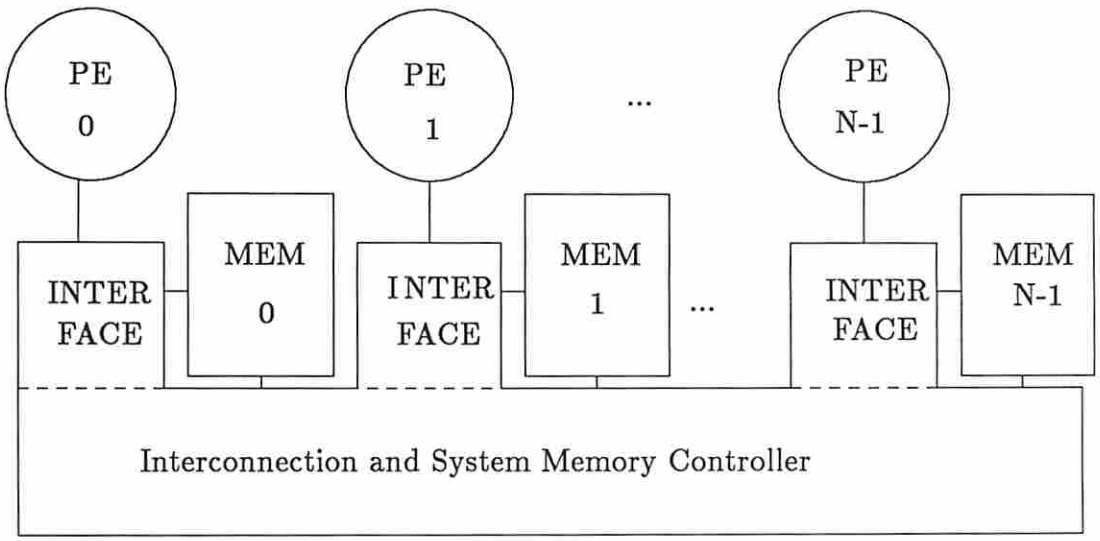


Figure 2.4: *Multiprocessor architecture with fully replicated memories.*

the local memory and the word must be propagated to all other remote memories. The interconnection and system memory controller must accomplish this in a manner which conforms to our desired memory model. To make the memory system strictly coherent the following procedure can be used.

1. A processor Px issues a write to memory location Y and blocks.

2. A bus in the interconnection causes all remote copies of memory location Y to be locked. Processors attempting to access Y must wait.

3. The new value of Y is propagated from processor to processor and updates first the local memory and then the remote memories. Each memory is visited exactly once in a random order.

4. Whenever a remote copy of Y is updated it is unlocked to local read accesses and remains locked to local write accesses.

5. After the update of Y has propagated to all remote memories the write–lock on all copies is removed.

6. Processor Px is unblocked and can issue another access.

In this scheme all writes to the same memory location are serialized by the bus access. The scheme follows the requirement of strict coherence to the letter because loads will always return the value defined by the most recent write.

In this interpretation of strict coherence, *time* is taken into consideration. However, since the relative speed with which each processor operates has never been taken into account the notion is inappropriate. A processor Px has no *timetable* with which it determines that memory location Y must *now* be updated so that another processor's load will return the "new" value. Likewise, the other processor has no *timetable* either which indicates that *now* is the time to perform a load of Y because Px has just updated the variable. It is not time which determines the correct instance when a load should return a "new" value, but, rather, it is the global order of reads and writes that must be correct.

Let us alter the data update mechanism of the previous example.

1. A processor Px issues a write to memory location Y and blocks.

2. A bus (or any other global serializing device) locks all memory locations of Y for write–accesses only. A processor can still read the old value of Y.

3. First the local memory is updated and then remote memories are updated one–by–one. Whenever a memory location Y is updated it becomes locked to both read and write accesses.

4. When all memories have been updated they all are unlocked by a single bus command.

5. Processor Px is unblocked and can issue another access.

This scheme logically accomplishes the exact same as the scheme which first locks all memory copies to both read and write accesses. All reads which were performed on memory location Y, returning the old content of Y, while Y was being updated throughout the system, can be said to have occurred *before* the write has taken place. The write on Y only *takes place* once *all* copies have

been updated. The fact that some processors may have attempted to access the new Y, but were prevented from doing so until all updates had propagated, does not affect the logical correctness of the scheme. We can say that the waiting processor(s) simply *slowed down* temporarily. Since no assumption about the speed of processors has been made, this reasoning is perfectly valid.

Depending on one's interpretation of the definition of strict coherence this scheme may or may not adhere to it. It depends on when, exactly, a write becomes the *latest write.*

In both of the above schemes the update due to a single write becomes available at the same logical instant to all processors. In the first scheme this instant is when all memory copies are originally locked and in the second scheme it is when all memory locations are unlocked. If there exists a system–wide instant in which a single write occurs it follows that *all* writes are system–wide ordered. If every write occurs at the same time system–wide, the memory system is strictly coherent.

Once again the notion of *time* is used. The notion of time can be removed from the requirement if all its ordering consequences are preserved. The result of this "time removal" is: "A memory system acts strictly coherent if all writes cause local memory updates in the same order." Collier comes to the same conclusion in [14]. We adopt his terminology and call such a memory system *write ordered.*

Definition 2.2 (Write Ordered Memory) *A memory system is write ordered if every copy of memory is updated due to writes in the exact same order.*

The principle of write ordering can be applied to the architecture of Figure 2.4. A write is accomplished in the following way.

1. Processor Px issues a write to memory location Y and then blocks.

2. The update of Y is propagated from memory–to–memory in a *fixed order.* All updates, independent of which processor caused them, observe this

same order. The local memory of Px is not necessarily the first memory in this order.

3. Updates of the same or different memory location(s) can never pass each other on their fixed route. Since all updates first update the same memory it is the order in which this memory is updated that must be maintained for all other memory copies.

4. Processors are always free to read their local memory.

5. After memory location Y has been propagated to all memory copies, processor Px is unblocked.

A write ordered memory system is indistinguishable from a strictly coherent memory system. A memory–history diagram can easily be used to examine shared memory during an execution even though multiple copies of memory exist if memory is write ordered because each copy is updated in the same order. Figure 2.5 depicts the memory–history diagram of three loads a,b, and c. Since the behavior of a write ordered memory system is indistinguishable from a strictly coherent memory system, the same rules for load–lines hold. The two loads b and c occur after words 3 and 4 have been updated, respectively. In the case of load b, word 3 has been updated but word 4 has not been updated yet. In the case of load c, word 4 has been updated. Since all writes must occur in the same order for all processors, load c must also reflect the updated value of word 3—which it does.

2.4.3 Word ordered memory

While the definition of a write ordered memory system successfully removes the notion of time, it does not remove the principle of serialization. Since all writes must propagate in the same order, beginning with the same memory, obviously serialization still occurs.

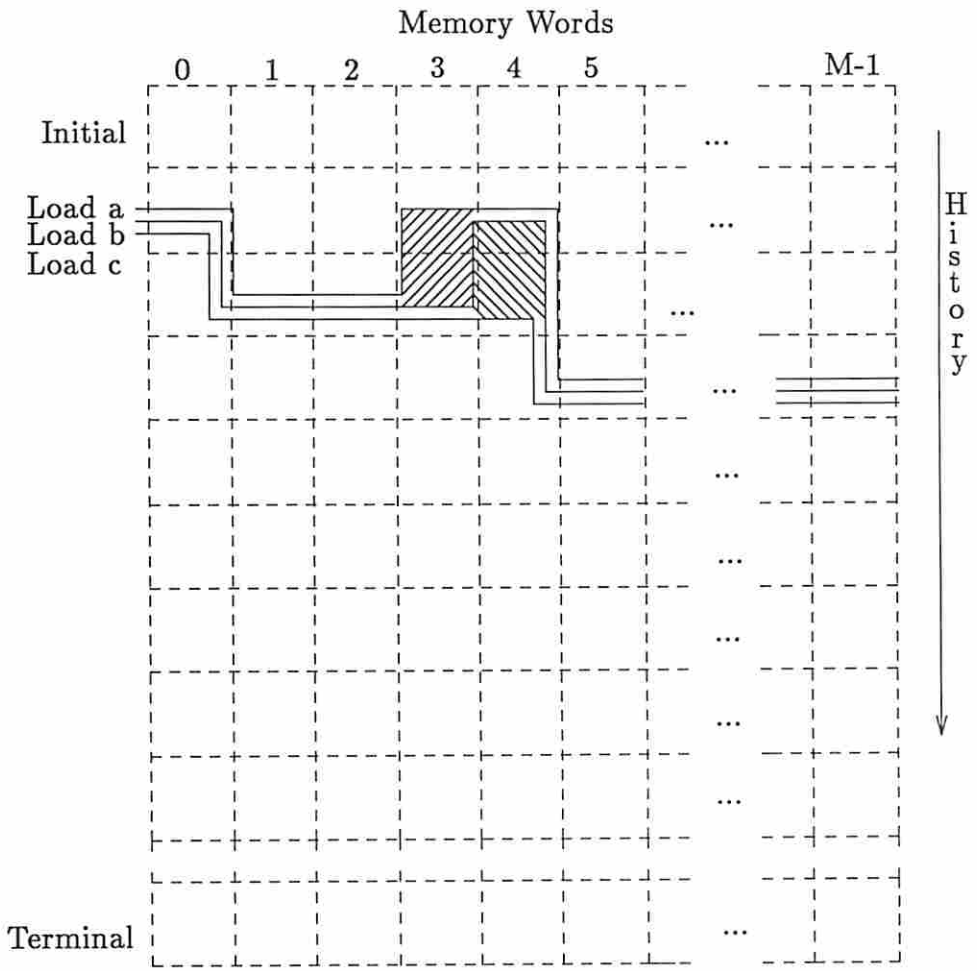


Figure 2.5: *Memory–history diagram of three loads.*

One result of the serialization of all writes is that copies of the same memory location ultimately will contain the same data. However, as is the case for the write ordered system, the different copies of memory can contain different data temporarily. If each processor stops issuing writes for some time, a write ordered system will "settle" and all memory copies will contain the same data. We call this property of a write ordered memory system *general coherence.*

Definition 2.3 (General Coherence) *A memory system is generally coherent if all copies of memory will contain the same data some finite time after all processors cease issuing memory write accesses.*

A system which is "somehow" forced to be generally coherent but not write ordered need not "funnel" all updates through a serialization device. In the case of a fully replicated memory, a write first causes the update of the local copy and then can begin propagating throughout the system. Unfortunately, such a memory system lacks an important quality, namely, *causal correctness.*

A write ordered memory system is causally correct. Causal correctness refers to the manner in which *cause* and *effect* can be observed via a memory system. In a shared memory multiprocessor a cause is a processor's state which makes it write to a memory location. The write's direct effect is the modification of a memory cell, or in the case of replicated memory, the modification of several memory cells—namely, all the copies of the addressed word throughout the system. If no processor ever reads the modified word, then the modification itself is the only effect of the write. However, if a processor reads the new value and the new value affects the processor and causes it to perform one or several writes which are a direct consequence of the original "effect" then all resulting writes are indirect effects of the original write. A write W2 is only an effect of another write W1 if the processor issuing W2 would not have issued the exact same W2 if W1 had not occurred.

The memory system has no means to detect whether a write is the indirect effect of another write. Therefore, it must assume that every memory

modification which has become available to a processor (i.e., all updates which have occurred at the processor's local memory) are the direct cause of all subsequent writes issued by the processor. If a processor's local memory has in the past been updated by a sequence of writes S_{in} and the processor itself in the future will issue a series of writes W_{out}, then it is the order in which other processors are affected by S_{in} and W_{out} which has direct impact on causal correctness.

Causal correctness is related to the manner in which writes cause update propagation throughout the system. In a causally correct system a processor must not be able to observe an effect (direct or indirect) before being able to observe the cause itself. Take the following sequence of events.

1. Processor Px performs a write on memory location Y.
2. Processor Py reads the value of Y.
3. Processor Py then performs a write on memory location X.
4. Processor Pz reads first X and then Y.

In a causally correct memory system if processor Pz reads the new value of X, then it cannot read the old value of Y. This the case because, the write performed by Py may have been the effect of observing the new Y. However, if processor Pz reads the new value of X it does not necessarily have to read the new value of Y defined by processor Px because, theoretically, another processor may have updated Y to an "even newer" value. This outcome is valid as long as Pz cannot read an old value of Y.

Definition 2.4 (Causal Correctness) *A shared memory system is causally correct if for any processor Px, data which has become available to Px before issuing a write Wx, has also become available to any processor Py before the data modified by Wx becomes available to Py.*[3]

[3] An interesting analogy can be drawn here. In a system with fully replicated memories, such as the one shown in Figure 2.4, if we assume that propagation speed is constant then delays can be equated with distance. A system with these qualities can be modeled in 3–D space if all propagations travel a straight–line path. A system which violates causal correctness *cannot* be modeled in such a 3–D space.

Let us assume that the architecture of Figure 2.4 uses a causally correct memory system. A processor may always read its local memory. Upon a write the processor blocks and memory copies are updated in some order which does not violate causal correctness. Since causal correctness does not enforce general coherence, let us further assume that the system is also generally coherent. Since concurrent writes to the same memory location by different processors can occur, the enforcement of general coherence does pose a problem. Time-stamping, processor prioritization, and utilizing fixed communication delays are some the possible solutions to this problem.

Take two concurrent write accesses to the same memory location by two processors. Both updates start propagating but one of the updates is not allowed to overwrite the other update while the other is allowed to do so. In the end all memory locations will contain the same data. For the example's sake let us assume that this is accomplished by time–stamping each write with the local time of issuance and an update always only overwrites an "older" copy of the word. Let us further assume that two processors cannot issue writes at the precisely same time—all time stamps are different.[4] For such a system we make the following claim.

Theorem 2.1 *In a generally coherent memory system each copy of a word is logically updated in the exact same order.*

In Theorem 2.1 the meaning of "logically updated" refers to the fact that while the order may actually not be the exact same, it is impossible for a program to detect this. This will become clearer in the following proof.

Proof: A write by a processor either will temporarily affect all copies of the addressed memory location (category 1) or will affect a (non–zero) subset of all copies of the memory location (it will at least temporarily affect its

[4] Generally coherent memories do not need to rely on real–time clocks for time stamping. Logical clocks, as proposed by Lamport in [41] can be used as well. However, in many architectures general coherence can be accomplished without clocks.

own copy) (category 2). Each memory copy will be affected by updates of category 1 in the same order because accesses of category 1 do not overlap. Take two subsequent updates of the same memory location and call them X and Y, both of category 1. Let the local memory of processor P_m experience a sequence of updates of category 2 which we call S_m which occurs after X and before Y. Processor P_m's memory is updated in the order X,S_m,Y. Similarly a processor P_n's memory is updated in an order X,S_n,Y.

Any two accesses a_p and a_q which are present in *both* S_m and S_n must also occur in the same relative order in both S_m and S_n because an update appears only in either series if it is "newer" than all previous updates in the series at the time of arrival (otherwise it is not allowed to modify memory). If a_p and a_q do indeed appear in both series then they will appear in both series in the same absolute (time) order in which they were issued by their respective processors. (If time-stamping is not used to determine order, then there still is some other type of mechanism which unambiguously orders accesses to the same word.) Otherwise one of the two updates would be "newer" than the other update in one processor while "older" than the other update in the other processor. This would violate general coherence which assigns order to the updates.

It is further true that any update a_p present in S_m cannot occur before X or after Y in processor P_n's memory because if a_p appears in S_m then it must be "newer" than X and "older" than Y. Since it is newer than X, the sequence a_p, X cannot occur in P_n's memory. Since it is older than Y, likewise, the sequence Y, a_p cannot occur in P_n's memory. The total number of updates present in both S_m and S_n may be different. For example, S_n may be lacking some updates present in S_m. For the sake of the proof we state that actually all updates in S_m are also present in S_n, but they were *overwritten so quickly* in processor P_n that processor P_n could never read them. This assumption does not affect the validity of the proof. Therefore,

every processor's memory experiences all updates to the same memory location in the exact same order. □

We call a memory system which is causally correct and generally coherent a *word ordered* memory system. Note, that in a word ordered memory system two processors can "observe" two updates of *different* words in a different order. This difference from a write ordered memory system can be detected by a program. For example, if a processor Px executes two loads, of A and B, and another processor Py executes two loads of B and A, processor Px can read new_A, old_B and Py can read new_B, old_A. When processors Px and Py compare results on the order of updates, the inconsistency becomes apparent.

In Figure 2.6 a memory–history diagram is shown for two loads, a and b. The two loads occur at times when with respect to one load memory word 2 has been updated and memory word 4 has not been updated while with respect to the other load, the opposite is true. This particular memory–history is possible in a generally coherent, causally correct shared memory system. In Figure 2.7, four loads by two different processors are depicted in a memory–history diagram. Loads a and b are subsequent loads by one processor and loads x and y are subsequent loads by another processor. A generally coherent, causally correct memory system *does not* allow the outcome of Figure 2.7 because the two processors observe the update of memory word 3 in different orders.

Theorem 2.2 *While the updates of different words in a word ordered memory system can be observed in different orders by different processors, this is possible only if the sequence of the updates is arbitrary to begin with.*

Proof: Either the order of updates of different memory locations by different processors is arbitrary or it is not.

If the order is not arbitrary then it must be deliberate. A deliberate sequence requires control. For example if write access X by processor Px must occur before write access Y by processor Py, then some entity must

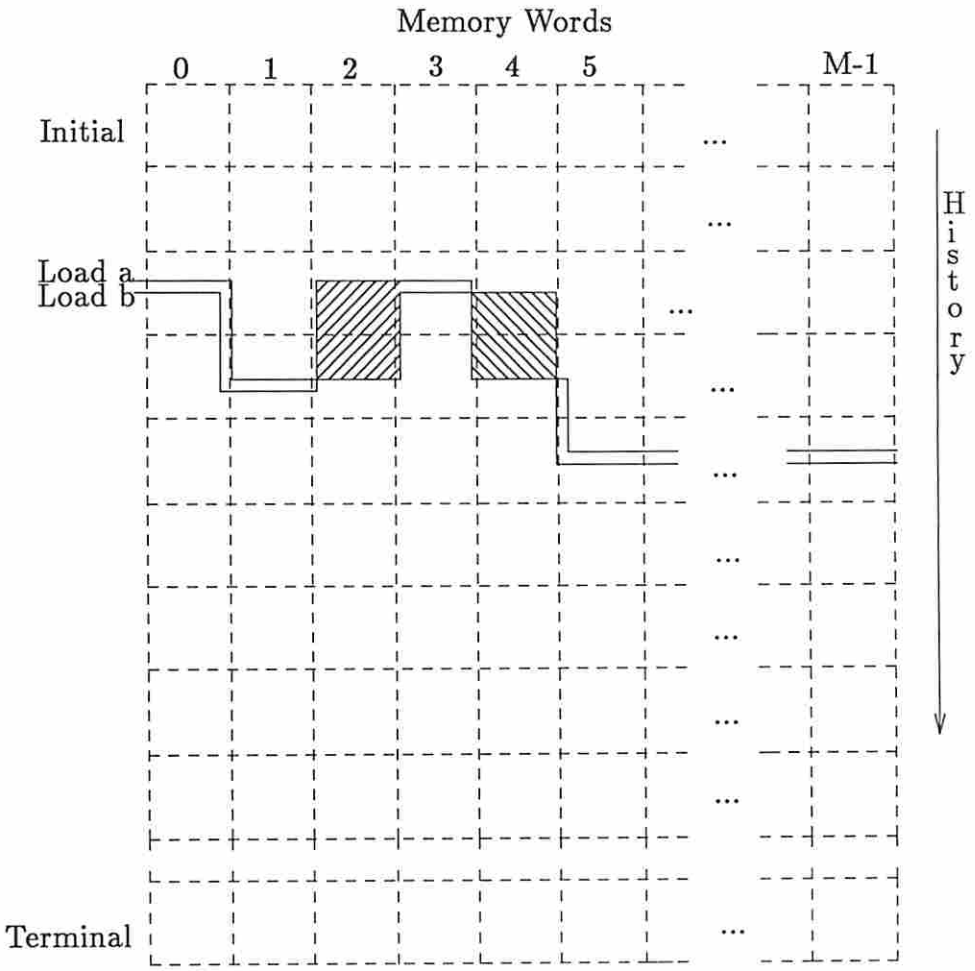


Figure 2.6: *Memory–history diagram of two loads.*

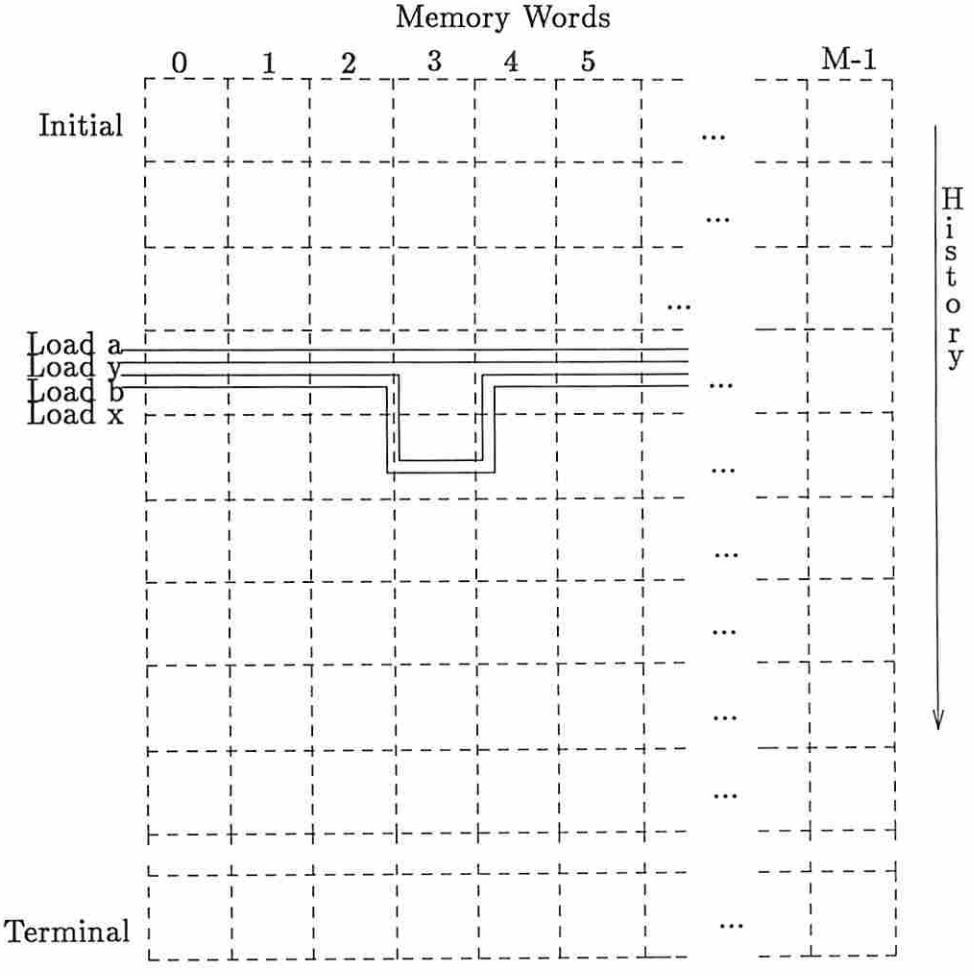


Figure 2.7: *Memory–history diagram of four loads by two processors. The outcome is* not *possible in a generally coherent, causally correct memory system.*

control this sequence. To enforce X before Y, Y must be blocked by the "controller" until the controller has become aware that X has taken place. The controller can either be Px (case 1), Py (case 2), an observer of the reversed events (case 3), or another processor (case 4).

1. In case 1, processor Px has to unblock processor Py when it is assured that X has taken place. This is the case after Px performs X. When Px has performed X, all that is certain is that Px's memory now reflects X (or an even newer value) because of the uniprocessor memory rule. Afterwards Px can send a "signal" to Py that Py can proceed with Y. This "signal" must be a write since all communications must be made via shared memory. The fastest way to get the signal to Py is to make X itself the signal. This means that Py unblocks itself after observing X. In this case no other processor can observe X and Y out of order, because Py observes X and then causes Y. A processor observing Y and then X violates the definition of causal correctness.

2. In case 2, processor Py has to first trigger X and then perform Y. The only way Py can cause Px to perform X is to send it a signal, i.e., perform a write. But Py can make no assumption on the speed with which Px will respond. Hence, Py must wait until it observes X before issuing Y. This is the same result as in case 1.

3. In case 3, a separate processor Pz both controls X and Y and then observes them out of order. It can only accomplish this by first triggering X and then triggering Y. But again, Pz can make no assumption about the relative speed with which the processors will react. Hence, the fastest way for it to assure X before Y is to send a signal to Px and wait until it observes X. Then it proceeds to trigger Y. But, in this case Pz *does not* observe X and Y out of order.

4. In case 4, a fourth processor Ps has to trigger X and Y. It must do so in the same way Pz does in the previous case. Let us call the

write which Ps performs to trigger processor Px, S1 and the write Ps performs to trigger Py, S2. Then X is an effect of S1, S2 is an effect of X, and Y is an effect of S2—but this also makes Y an effect of X. In a causally correct memory system Pz cannot observe an effect before observing the cause. Hence, Pz must observe X before Y.

Therefore, two writes cannot be observed in different orders if that occurrence is not arbitrary. It follows that no number of writes can be observed out of order if their order is not arbitrary. □

In many shared memory synchronization protocols two events, i.e., writes, can occur in an arbitrary order (i.e., a race condition) but the actual order of occurrence *does* have an effect on the future of the involved processors. For example, one processor is allowed to access a critical section while other competing processors are blocked (busy waiting). Processors which are "active participants" in a race condition will all observe the relevant updates of different memory locations in the correct order because algorithms which are based on the outcome of arbitrary sequences are inherently founded on concurrency model assumptions which require strict sequencing of accesses. To enforce strict sequencing of accesses other restrictions must be enforced which result in uniform observation-order of memory updates when such algorithms are executed. In the Section which discusses this concurrency model it becomes clear why these algorithms do function correctly in causally correct, generally coherent systems.

2.4.4 Non-ordered memory

A further relaxation of the rules which govern memory behavior is possible. *Non-ordered* memory is not causally correct and may or may not be generally coherent. If Figure 2.4 depicts a system with non–ordered memory then each processor is *usually* allowed to read and write its local memory. Writes to a local memory propagate in a random order updating remote memories. A processor

can proceed to access memory as soon as its local copy has been updated after a write. (The uniprocessor memory rule is observed.)

A processor can be in one of two states. (1) Either *all* the writes which it has issued in the past have propagated to *all* remote memories, or (2) some writes it has issued in the past have not propagated to all other remote memories yet. In the first case we say that the processor's accesses *have been globally performed.*[5] Besides the non–ordered portion of memory a non-ordered memory system also has a secondary memory which is either strictly coherent, write ordered, word ordered, and/or capable of executing indivisible read-modify-write memory accesses. In the context of a non–ordered memory system, we call this secondary memory the *synchronization memory.* Accesses to synchronization memory are serviced in a manner consistent with the synchronization memory's memory model—except, that all synchronization memory accesses by a processor Px are blocked until all non-synchronization memory writes issued by Px have been globally performed.

2.5 Parallel Programs and Synchronization

As has been shown in Section 2.1, a multiprocessor can execute even a trivial parallel program in many different ways resulting in many different outcomes. The program of Section 2.1 has different final states, even if processors are strictly sequenced and if the memory system is write ordered. If this concurrency model is assumed then the outcome 111111 is possible, for example. Several different *execution paths* which adhere to the concurrency model can yield this result. Hence, even in this restricted concurrency model several solutions are correct and each solution may have several execution paths which lead to it.

A useful concurrency model restricts the possible event orderings. Using the understanding of the concurrency model and programming constructs the programmer wishes to devise a program which correctly executes a parallel algo-

[5]This term will be more rigorously defined in Section 2.8.

rithm. The parallel program itself further restricts the possible paths that can be taken by the execution but does not usually restrict it to a single one—several paths can yield the same one correct result(s). There exists a class of parallel algorithms, called *asynchronous algorithms*, which do not require synchronization. We will ignore these types of algorithms in this thesis. All parallel *synchronous* algorithms must synchronize; otherwise they constitute independent sequential algorithms.

Processes must synchronize in order to enforce mutual exclusion during the execution of operations on shared writable data (i.e., *critical section*) or to coordinate process execution (i.e., *barrier synchronization*) [3]. Synchronization is accomplished by communication and conditional execution. All parts (processors) affected by a synchronization must cooperate. The concurrency model defines how communication occurs and hence affects the manner in which synchronization can be implemented.

2.5.1 Time-shared uniprocessors and sequential consistency

Multiprocessing can be implemented in time-shared uniprocessors. In a time-shared uniprocessor, only one process runs at a time but "slices" of each process execution are executed in turn. In this case, communication and synchronization protocols among processes can be implemented with simple loads and stores of shared data. A trivial synchronization algorithm using only loads and stores is shown in Figure 2.8 for two processes. (The fact that this algorithm can lead to deadlock is orthogonal to this discussion.) An extension of this algorithm to enforce mutual exclusion of N processes, using simple loads and stores was given by Lamport [39]; it is shown in Figure 2.9.

Lamport's "bakery" algorithm relies on processes "taking a number" and waiting for their turn to enter the critical section. If two processes have the

```
--------------------------------------------------------------------
        ********PROCESS 1:********
               A:=1;                    /event S1(A)
        L1:    If (B=1) then goto L1;   /event L1(B)
               <critical section>
               A:=0;

        ********PROCESS 2:********
               B:=1;                    /event S2(B)
        L2:    If (A=1) then goto L2;   /event L2(A)
               <critical section>
               B:=0;
--------------------------------------------------------------------
```

Figure 2.8: *Synchronization protocol using two shared variables A and B. Initially A=B=0.*

```
-----------------------------------------------------------------
          begin integer j;
      L1:   choosing[i]:=1;
            number[i]:=1+max(number[1],...,number[N]);
            choosing[i]:=0;
            for j:=1 to N do
            begin
      L2:     if choosing[j] <> 0 then goto L2;
      L3:     if number[j] <> 0 and
              ((number[j],j) < (number[i],i)) then goto L3;
            end;
            <critical section>
            number[i]:=0;
            <non-critical section>
            goto L1
            end;
-----------------------------------------------------------------
```

Figure 2.9: *N-process mutual exclusion algorithm (Lamport '74), shown for process i. The relation (a,b) < (c,d) is true if a < c, or if a=c and b < d.*

same number, then the process with the lower process ID has priority. Similar algorithms have been derived by Dijkstra [17], Knuth [35], and others.

Figure 2.10 shows a barrier synchronization algorithm for N processes operating concurrently on the *same* iteration of a loop. Before a process is allowed to start processing the next iteration, all processes must have finished the present iteration. One master process N controls the barrier, and N-1 slave processes cooperate with the master process.

Figure 2.11 shows a simple producer-consumer algorithm using a limited buffer of size b [40]. The producer may not overwrite data in the buffer or attempt to write to a full buffer. The consumer reads a data item only once, and may not attempt to read from an empty buffer. Many other communication and synchronization algorithms relying on simple loads and stores of shared data could be designed by individual programmers. In order for a multiprocessor to correctly execute these algorithms it must adhere to a concurrency model which forces a multiprocessor to execute programs in such a way that the outcome is equivalent to one generated by a multitasking uniprocessor. This concurrency model is called *sequential consistency*, and was defined by Lamport in [42].

Definition 2.5 (Sequential Consistency) *[A system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by its program.*

The model of behavior of a multitasking uniprocessor and a sequentially consistent multiprocessor are the same. It should not be possible to determine by means of software only whether a program was executed on a multitasking uniprocessor or on a sequentially consistent multiprocessor.

2.5.2 Concurrent consistency

Sequential consistency is very restrictive in the way it enforces event ordering. This will become more obvious when, later in this Chapter, the condi-

```
--------------------------------------------------------------------

        ********MASTER PROCESS N********

             for i= 1 to k do
             begin
               do_iteration(N,i);
               for j = 1 to N-1 do
         L1:     if flag[j] = 0 then goto L1;
               for j = 1 to N-1 do flag[j] := 0;
             end;

        ********SLAVE PROCESS p********

             for i = 1 to k do
             begin
               do_iteration(p,i);
               flag[p] := 1;
         L2:   if flag[p] = 1 then goto L2;
             end;
--------------------------------------------------------------------
```

Figure 2.10: *N-process barrier synchronization. All processes concurrently work on iteration i and may not proceed to iteration i+1 until all processes are finished with iteration i. Initially all flags are set to 0.*

```
--------------------------------------------------------------------

        ********PRODUCER*********

L1:     if s-r mod k = b then goto L1;
        <put message into buffer>
        s:= s+1 mod k;
        goto L1;

        ********CONSUMER*********

L2:     if s-r mod k = 0;
        <take message from buffer>
        r:=r+1 mod k;
        goto L2;
--------------------------------------------------------------------
```

Figure 2.11: *A producer-consumer algorithm by Lamport (Lamport '77).*

tions for sequential consistency will be defined. There exists another concurrency model, besides sequential consistency, which allows the correct parallel execution of "sequential programs." We call this new concurrency model *concurrent consistency.* A concurrently consistent multiprocessor *does not* model a sequential system but rather, presents a parallel environment in which the exact time at which write accesses are made by individual processors can be globally ordered with respect to all other writes but in which data due to writes can become available at different times for different processors. That is, the event itself (the write) is globally ordered, but its consequence (i.e., the modification of memories) cannot be globally ordered. The order in which writes *do become* available is restricted such that synchronization algorithms as shown in Figures 2.7–2.11 function correctly. There do, however, exist programs which result in different executions on sequentially and concurrently consistent multiprocessors.

Definition 2.6 (Concurrent Consistency) *A system is concurrently consistent if it executes all programs in the way that sequentially consistent systems do except for programs which explicitly test for sequential consistency or take access timings into consideration.*

A program can test for sequential consistency by allowing two or more processors to compare the orders in which memory updates have been observed. Such a program can detect the difference between a sequentially and a concurrently consistent system. Except for this purpose, such a program is not useful because in a sequentially consistent environment such a test serves no purpose.

It was shown in Section 2.4.3 that two or more writes by different processors, if controlled in sequence in any way, will be observed in the correct order by all processors. If two or more writes occur in an arbitrary order, i.e., as the result of a race condition, then these writes can be observed in different orders by different processors, which do not themselves participate in the race condition. The question remains whether such observations can lead to "faulty" conclusions by a processor.

If programs do not take timing into consideration (as we generally assume) then no conclusions can be drawn from the observation of two or more writes in different orders. Take, for example, the synchronization algorithm of Figure 2.8. and assume that each process is designed to ensure the mutual exclusion of two *different* processes, residing on different processors. Assume that process 1 acts as a guardian process of process 3 and process 2 acts as a guardian process of process 4—all four processes reside on different processors. Processes 3 and 4 do not actively participate in the algorithm but simply observe variables A and B and *deduce* when it is allowed to enter the critical section. Process 3 can spin on variables A and B which are both initially 0. Likewise, process 4 spins on the same variables. It is possible for process 3 to observe A=1, B=0 and process 4 to observe B=1, A=0. However, in this case, process 3 can only conclude that either (1) it has obtained its critical section *or* that (2) deadlock has occurred. The same conclusion can be drawn by process 4. In the case cited it is indeed true that deadlock will occur. A process cannot conclude that it has access to its critical section because while it is aware that variable A has been set (in the case of process 3) it cannot make any assumptions on *how soon* process 1 will execute the "if then on B" statement.

The programs of Figures 2.9–2.11 will also execute correctly on a concurrently consistent multiprocessor because processors do not explicitly test for sequential consistency (they do not compare results) and the algorithms are not based on timing (they are based on ordering). A program which does compare observation orders by different processors can conclude that two writes are *concurrent* if two processors observe them in different orders. This definition of concurrency is consistent with the one used by Lamport in [43] in the context of message-passing systems.

2.5.3 Hardware synchronization and communication primitives

A sequentially or concurrently consistent multiprocessor can implement all necessary communication and synchronization by using simple loads and stores. However, implementing synchronization and communication using shared memory loads and stores can be complex and error–prone. Debugging such algorithms is also very difficult. In some cases synchronization by "spinning" on a global variable can be inefficient if the variable is subject to a cache coherence protocol which causes the block the variable is located in to "ping–pong" back and forth between two or more caches [33].

There exists another class of accesses which are not simple loads and stores, they are *read–modify–write* accesses. A read–modify–write access first performs a read of a datum and then indivisibly modifies it. The modification is indivisible, because no other access to the datum is allowed (possible) after reading it and before writing it. This type of access does not fit into the memory models discussed so far. The implementation of indivisible read–modify–write accesses in a shared memory system which allows multiple copies of data to exist must be addressed individually for any given architecture. We assume, for the time being, that memory locations to which indivisible read–modify–write accesses are possible are not replicated. In the next Chapter an example is given which shows that this is not necessary in all architectures.

A common example of an indivisible read–modify–write primitive is the *test&set* instruction. A test&set instruction returns the value of a binary variable before it is set to 1. For example, with a simple *test&set* primitive, the critical sections of Figures 2.8 and 2.9 can be implemented by the code, shown in Figure 2.12 [22]. This code is simpler than the code using loads and stores only.

The program for the communication channel of Figure 2.11 can be rewritten as shown in Figure 2.13 to utilize the test&set primitive. Other implementations are possible, and more sophisticated primitives than test&set may

```
L1:  If test&set(x) = 1 then goto L1;
     <critical section>
     reset(x);
```

Figure 2.12: *Mutual exclusion algorithm based on a test&set primitive.*

```
     ********PRODUCER********
L1:  If (test&set(x) = 1) then goto L1;
     If (s-r mod k = b) then {reset(x); goto L1;}
     <put message into buffer>
     s:= s+1 mod k;
     reset(x);
     goto L1;

     ********CONSUMER********
L2:  If (test&set(x) = 1) then goto L2;
     If (s-r mod k = 0) then {reset (x); goto L2;};
     <take message from buffer>
     r:=r+1 mod k;
     reset(x)
     goto L2;
```

Figure 2.13: *Producer-consumer algorithm using the Test&set primitive.*

exist in any given multiprocessor. However, the need for indivisible read-modify-write operations—such as test&set—poses problems to the hardware. Either memory locations must be locked, and access to them is denied to all processors except to the one executing the read-modify-write operation, or the memory system itself must *execute* the entire read-modify-write operation indivisibly.

On the other hand, when indivisible instructions are not used to implement synchronization algorithms, then the multiprocessor must be sequentially or concurrently consistent. Depending on the underlying architecture of the multiprocessor, this restriction can be difficult to enforce. The complexities and efficiencies of the programs written with and without special synchronization primitives are difficult to compare and probably depend on the specific problem. Ideally, multiprocessor systems should support a set of flexible synchronization primitives and be sequentially or concurrently consistent as well.

2.5.4 Weakly ordered systems

To be able to synchronize using simple loads and stores a multiprocessor must be either sequentially or concurrently consistent or must be able to implement indivisible read–modify–write memory accesses. The exact requirements on the processor and memory model of such systems are stringent, as will be shown in Section 2.9.

In a *weakly ordered* concurrency model, simple loads and stores cannot be used to implement synchronization and communication. Special hardware–recognized synchronization accesses must be used to accomplish these tasks. A weakly ordered system may *not* correctly execute the programs of Figures 2.8–2.11. In a weakly ordered system processors must synchronize with explicit synchronization primitives to access shared writable data. Code protected by these synchronization primitives is either a critical or a semi-critical section. Note that weakly ordered systems do *not* need to be generally coherent per se—general coherence results only from correct programming of such systems.

2.6 Buffering, Caching, and Combining

In high-end processors, the pipelining of instruction execution is common place and is aided by extensive prefetching and buffering of memory accesses. These mechanisms can have direct impact on the processor and memory models. With respect to all data any processor which is part of a multiprocessor architecture must adhere to the uniprocessor rules. Usually a processor can distinguish between shared writable and private data. After making a shared writable data access, a processor can make any number of private data accesses without regard to the processor or memory model as long as the shared data access conforms to the uniprocessor rules. It is only when a subsequent shared data access is to be made that all multiprocessor rule pertaining to the model must be upheld.

2.6.1 Memory access buffering techniques

A scheme supporting operand prefetching is implemented in the IBM3033 by the operand registers [15]. Up to six operand accesses may be in progress at the same time. A similar procedure is implemented for the stores. Stores to memory occur later in the pipeline, when the execution of an instruction is completed. Usually, stores can be completed in one pipeline cycle, by simply writing data and addresses to an operand store buffer. Likewise, processors of the IBM 3090 multiprocessor system prefetch both instructions and data while maintaining multiple instructions streams to accommodate all possible targets of a branch instruction [64]. Dependency checking among the different instruction streams (which all belong to the same sequential task) is also done in this system to ensure "hazard-free" sequencing. Buffering stores is particularly efficient because the processor does not need to wait for the return of information from memory, contrary to the load of an operand. As a result, loads are more critical for performance and are often given higher priority over store requests.

In a multiprocessor system, memory accesses can be buffered in the processor, in the interconnection network, and in the shared memory. Several

paths may be possible between a processor and a memory module. If invalidations or stores have to modify an entry in a private memory or cache of another processor, these updates may be buffered in the destination processor node. A good example is the BIAS filter in the IBM3033 which stores and selectively filters invalidation signals coming from other caches [10]. (Systems with dual cache directories may be analyzed as systems with single invalidation buffers, since a lookup of the dual directory introduces a delay of at least one cycle between the bus and the cache.) Buffering instruction fetches is safe for a pipelined machine in a multiprocessor if instructions are not modifiable. Self-modifying code poses special problems which will not be addressed in this thesis. *Buffer management* refers to the order in which multiple buffered requests are treated. In most cases, the requests are served in a strict FIFO order (First-In-First-Out). In some cases requests may be allowed to pass each other in the buffer. This is referred to as *jockeying.* Jockeying is often permitted among memory requests for different memory words, but it is not permitted between requests destined to the same memory word. Jockeying with this restriction is called *restricted jockeying.*

2.6.2 Caching techniques

Memory caching is an effective method to reduce both the average memory access latency and contention. In multiprocessors, caching of shared writable data causes the well-known multiprocessor cache coherence problem. This problem has been addressed by many researchers [18,5]. In the following chapter, multiprocessor cache coherence is defined in the context of sequentially consistent multiprocessors.

To support a processor with extensive operand buffering, accesses to the cache are usually pipelined and the cache may be lockup-free [37]. A lockup-free cache does not block (or lock up) the processor on a miss. Rather, it records the status of the memory request causing the miss and keeps accepting and servicing requests from the processor. Lockup–free caches in multiprocessors are analyzed in Chapter four.

2.6.3 Combining interconnections

Combining interconnection networks are capable of *combining* a load with a store to the same memory location within the interconnection [31]. The load can be completed before it propagates all the way to the memory. As a result, both load latencies and memory contention are reduced. Combining networks may cause logical problems, since concurrent load operations may return different values. Combining networks act like memory and must therefore adhere to the system's memory model.

2.7 Inter–process Dependencies

Whether a concurrency model requires strict or hazard–free sequencing depends on whether the concurrency model preserves *inter–process* dependencies. Sequential and concurrent consistency both *do* preserve inter–process dependencies and therefore require strict sequencing. A weakly ordered system does not preserve inter-process dependencies and therefore hazard–free sequencing can be used in such systems.

Take for example the simple synchronization algorithm of Figure 2.8. In the first processor's code there are two relevant shared data accesses. The access S1(A) stores a value in variable A, and the access L1(B) tests variable B (which implies loading B). There exist no hazards between the two accesses because they refer to different memory locations. Therefore, a uniprocessor could execute them out of order or at least pipeline the execution and end up performing the load before the store. The same is true for the mirror–image code executed by the second processor. If both processors execute their respective data accesses out of order then the algorithm fails and both processors will simultaneously enter the critical section.

Note that the same out–of–order execution of accesses is theoretically possible in a multitasking uniprocessor. However, the problem is only the same if the processor context–switches in the middle of executing two accesses out of

Ordered Interleavings Results:

$S1(A) \rightarrow L1(B) \rightarrow S2(B) \rightarrow L2(B)$ Processor 1 enters CS.
$S2(B) \rightarrow L2(A) \rightarrow S1(A) \rightarrow L1(B)$ Processor 2 enters CS.
$S1(A) \rightarrow S2(B) \rightarrow L1(A) \rightarrow L2(B)$ Deadlock.
$S1(A) \rightarrow S2(B) \rightarrow L2(B) \rightarrow L1(A)$ Deadlock.
$S2(B) \rightarrow S1(A) \rightarrow L1(A) \rightarrow L2(B)$ Deadlock.
$S2(B) \rightarrow S1(A) \rightarrow L2(B) \rightarrow L1(A)$ Deadlock.

Figure 2.14: *All allowable interleavings of stores and loads for Figure 2.8.*

order. This is generally not allowed in a uniprocessor because of the complexity of the state which the processor would need to save before context switching. For example, if the reverse execution of the accesses is the result of pipelining, a processor will allow the pipeline to empty before context switching. Figure 2.14 shows the only *allowable* execution interleavings of the algorithm of Figure 2.8. In this case "allowable" refers to the only interleavings possible for which the algorithm functions correctly.

An inter–process dependency (or *inter–dependency*) cannot be detected locally by a processor. Sequentially and concurrently consistent multiprocessors therefore must issue all shared data accesses as if there exist inter-dependencies on the data. Issuing reads and writes in reverse program–order is not the only way inter-dependencies can be violated.

For other types of synchronization and communication protocols the effects of non-sequentially consistent multiprocessors can be more subtle. Take for example the barrier protocol of Figure 2.10. Apparently slave processes cannot overrun the synchronization barrier since an intra-process (hazard) dependency

```
-------------------------------------------------------------
        Processor 1              Processor 2
      ---------------------------------------
        a: M(x):=data;           c: reg1:=M(y);
        b: M(y):=1;              d: reg2:=M(x);
                                 e: if reg1<>1 then goto c;
                                 f: use_data(reg2);
-------------------------------------------------------------
```

Figure 2.15: *Two program segments to be executed concurrently.*

does exist. This is due to the fact that processes block *themselves* by setting flag[p] equal to 1 after an iteration and then spin on the flag. However, if the loop is viewed as an unraveled sequence of *do_iteration(p,i)* procedures, separated by barrier synchronizations, then it becomes apparent that the synchronization points do not prevent the process from "entering" the next iteration if, for example, the processor prefetches operands. This is the case since no intra-process dependency exists between the loop which tests flag[p] and the next iteration of *do_iteration(p,i)*. Consequently, the barrier synchronization of Figure 2.10 might not function correctly in a non-sequentially or non-concurrently consistent multiprocessor.

It is not only the out–of–order execution of loads and stores that can violate intra–process dependencies. Exchanging the order of execution of two loads within a single process can have the same effect. The parallel program segments in Figure 2.15 demonstrate this. Memory location x contains data generated by processor 1. Memory location y is used as a flag to signal to processor 2 that the data in x are valid. Processor 2 spins on the flag which should be zero initially. If statements (c) and (d) of processor 2 are executed out of program order then in the case of an actual interleaving of (d,a,b,c,e,f) processor 2 will not use the data passed by processor 1 via memory location

x. Statements (c) and (d) are both simple loads and their reversal violates an inter-dependency.

A programmer, having used the synchronization protocols defined in Figures 2.8–2.11 assumes the concurrency model of sequential or concurrent consistency, and therefore such a multiprocessor must only allow ordered interleavings of events. Since inter-process dependencies are caused by accesses of shared writable data (we ignore interrupts), the memory model used in a sequentially or concurrently consistent multiprocessor is also important.

The memory models defined in a previous Section used the terms *load, store, issuing an access, performing an access*, and *updating a replicated memory copy.* In the presence of buffering of accesses and caches the terminology must be refined.

2.8 Load and Store Events in Multiprocessor Systems

When processors and memories of a multiprocessor are modeled as simple devices interacting by using such simple commands as *read(address), write(address)(data), send(data),* and *update(address)(data)* then proving conformance to a processor or memory model is simple. However, in real architectures a multitude of signals exist, memory accesses must be at least partially overlapped, and accesses can be in various states of completion. In cache–based systems and in systems using combining networks the exact physical location at which data will be retrieved can be unknown at the time of the access. It is necessary to properly define the state a memory access is in, such that the terminology is architecture and timing independent.

Definition 2.7 (Memory Access States—Unique Copies) *A memory access request is initiated when a processor has sent the request and the completion of the request is out of its control. An initiated request is issued when it has left*

the processor environment[6] *and is in transit in the memory system. A load is considered performed at a point in time when the issuing of a store to the same address cannot affect the value returned by the load. A store on X by processor i is considered performed at a point in time when a subsequently issued load to the same address returns the value defined by a store in the sequence* $\{Si(X)\}+$.

In this definition, we denote load and store accesses by processor i on variable X as Li(X) and Si(X), respectively. In a system where memory store operations become "observable" at the same time for all processors, the store events on one variable can be ordered based on the physical time at which they are performed. The notation {Si(X)}+ is used to denote the sequence of stores on X following the store Si(X) and including Si(X) in the ordering based on the time when the stores are performed. Similarly, the notation {Si(X)}- denotes the sequence of stores on X preceding Si(X) but not including Si(X).

For example, in the system depicted in Figure 2.19 operand prefetching is implemented through a prefetch buffer at the processor. The processor *initiates* the operand prefetch by placing the operand's address in that buffer. Then the buffer controller *issues* the operand fetch to the shared interconnection and memory. The request transits in the interconnection and it is *performed* when it is latched in a buffer associated with the memory, provided this buffer is FIFO (restricted jockeying is allowed in the memory buffer). The situation is similar for a store request. The processor *initiates* the store by placing the request in a store buffer at the processor. Later on, the store buffer controller *issues* the store request to the interconnection. The request is then in transit in the interconnection. It will be *performed* as soon as it is latched in the FIFO buffer associated with the destination memory.

Definition 2.7 is relevant whether or not multiple copies of the same data exist in the system. When multiple copies of data do exist, such as in cache-based systems (but not only limited to cache-based systems), we define the notion of *performed with respect to a processor.*

[6]The processor environment includes the CPU and local buffers.

Definition 2.8 (Memory Access States—Multiple Copies) *A store by processor i is considered performed with respect to processor k, at a point in time when a subsequently issued load to the same address by processor k returns the value defined by a store in the sequence {Si(X)/k}+. Similarly, a load by processor i is considered performed with respect to processor k, at a point in time when a subsequently issued store to the same address by processor k cannot affect the value returned by the load.*

The definitions of {Si(X)/k}+ and of {Si(X)/k}- are similar to the definitions of {Si(X)}+ and {Si(X)}-. However, the stores on X are ordered according to the time when they are performed with respect to processor k.

From the above definition it should now be clear why we consider some events not to occur at instances in time, but rather relative to each other. In some architectures, for example, at time t a store may have been performed with respect to processor x but not yet with respect to processor y. It is certainly futile to attempt to associate a particular point in time with the store event. Note that the notion of *performed* refers to the mutability of a subsequent event (i.e., after a write is performed w.r.t. processor P, a read by processor P *cannot* reflect anymore the value valid previous to the write). It may be difficult for a processor to be able to pinpoint the exact moment when an access is performed with respect to another processor.

In systems where accesses are performed at different times with respect to different processors, we can define a point in time when an access becomes *performed with respect to all processors.*

Definition 2.9 (Globally Performed Accesses) *A store is globally performed when it is performed with respect to all processors. A load is globally performed when it is performed with respect to all processors and when the store which is the source of the returned value has been globally performed.*

Once a store is globally performed, no issued load by any processor can return an old value which was valid before the store took place. There is a subtle

difference between a load *performed with respect to all processors* and a *globally performed* load. Once a load is performed with respect to all processors, the value it returns is fixed and cannot be altered, independent of any action by any processor. When a load is globally performed it is additionally true that any other load issued subsequently by any processor cannot return a word which is "older" (i.e., a word in {Si(X)}-) than the word returned by the performed load.

2.9 Enforcing Concurrency Models

The necessary hardware behavior to enforce a given concurrency model can be derived by investigating the possible execution outcomes of some parallel program segments.

2.9.1 Sequential consistency

Strict sequencing of memory accesses is a prerequisite of sequential consistency. In a system with strictly coherent memory, it is easy to see that accesses are strictly sequenced by simply performing them in program order. That is, to guarantee strict sequencing of accesses in such a system, accesses must be initiated and issued so that they will be performed in order. This is true because before the access has been performed, it can be considered as "private" to the processor that issued it. Immediately after it is performed, the store affects all other processors and the load ceases to be affected by all other processors, at the same time. The only problem is therefore to ensure that successive accesses from the same processor affect or are affected by all other processors in program order. It can generally be assumed that it is simple to control the sequencing of accesses up to the issuing level. Once an access is issued it must not be possible for a subsequent access by the same processor to "pass it."

Condition 2.1 (Strict Sequencing and Strictly Coherent Memories) *Strict sequencing of memory accesses is preserved in multiprocessors with strictly coherent memory if all accesses by a given processor are* performed *in program order.*

It is easy to see that condition 2.1 enforces sequential consistency but the question remains whether it is necessary. In Figure 2.16, three program fragments are given. They are sequences of loads and stores on different variables. For each fragment a specific condition for sequential consistency is given. It can be seen that if any two accesses belonging to the same process are executed out of order, then this condition may be violated.

A distinct problem is the possibility that a processor bypasses the memory system for successive accesses to the *same* variable, a technique called short–circuiting in [36]. A load on a variable directly following a store on the same variable can bypass the memory, because the resulting interleaving is ordered. However, the load cannot bypass the memory system if a load for a different variable is initiated by the processor between the store and the load. In essence, condition 2.1 is violated if short-circuiting is allowed. However, a multiprocessor which allows short-circuiting does not constitute a system with strictly coherent memory, per se. Consequently, the conditions defined in the next Section should be applied.

Strict coherence was defined in the context of memory systems which maintain only a single copy of each stored word. Such memory systems are *access atomic*. That is, a write, performed with respect to *any* processor is also globally performed. Some memory systems are access atomic even if multiple copies of a word can exist—in this case one or several buses must connect all memory modules. Other memory systems are not access atomic. Strict coherence in a system which can contain multiple copies of a memory word requires serialization enforcement. Condition 2.1 is not sufficient to enforce strict sequencing and hence sequential consistency in a system where multiple copies of memory exist without a serialization and/or broadcast device. Take for example the program fragment

---

FRAGMENT 1:

PE1:	PE2:
S1(A)	S2(B)
L1(B)	L2(A)

If L1(B) returns a value in {S2(B)}- then L2(A) must return a value in {S1(A)}+.

FRAGMENT 2:

PE1:	PE2:
L1(A)	L2(B)
S1(B)	S2(A)

If L1(A) returns a value in {S2(A)}+ then L2(B) must return a value in {S1(B)}-.

FRAGMENT 3:

PE1:	PE2:
S1(A)	L2(B)
S1(B)	L2(A)

If L2(B) returns a value in {S1(B)}+ then L2(A) must return a value in {S1(A)}+.

Figure 2.16: *Behavior of three program fragments in a sequentially consistent system.*

Processor 1	Processor 2	Processor 3
.	.	.
.	.	.
S1(A)	L2(A)	L3(B)
.	S2(B)	L3(A)
.	.	.
.	.	.

Figure 2.17: *Three processes sharing variables A and B.*

of Figure 2.17 further illustrated in Figure 2.18. In this program, if L2(A) reads the value produced by S1(A), and if L3(B) returns the value produced by S2(B), then L3(A) must also return the value produced by statement S1(A). However, if, for some reason, event S1(A) takes much more time to propagate to processor P2 than it does to processor P3, then there may be enough time (depending on conflicts and distances) for P2 to perform event L2(A), then S2(B), and for P3 to perform L3(A) on a value of A previous to S1(A). This is not an ordered interleaving of accesses.

In a system with strictly coherent memory the above sequence of events is not possible. A memory model with multiple memory copies but indistinguishable from a strictly coherent memory system was described in Section 2.4.2—it was called a *write ordered* memory system. Hence, the most general requirement of a sequentially consistent multiprocessor is that it employs *strictly sequencing* processors and a *write ordered memory* system.

Condition 2.2 (Sequential Consistency) *A multiprocessor is sequentially consistent if accesses by processors are performed in program order with respect to all processors and if the order in which all writes are performed is the same with respect to all processors.*

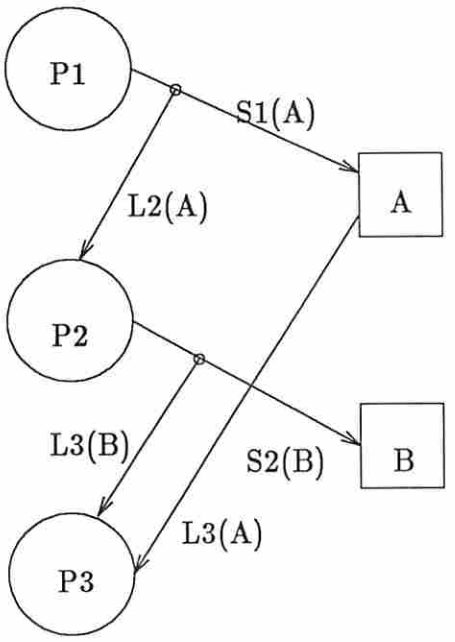


Figure 2.18: *Possible outcome of concurrent execution of programs in Figure 2.17.*

Condition 2.2 does not prevent a processor from issuing an access before a previous access has been globally performed. In most architectures meeting the required ordering restrictions of condition 2.2 is difficult if this is allowed. An easier to apply condition of sequential consistency is that all accesses must be globally performed in program order and that a processor cannot issue an (shared writable memory) access until the most recent shared writable memory access by the processor has been globally performed. Note that this condition does not allow a processor to continue with another (shared writable memory) access after a read returns a word which has been recently updated but which has not yet become available to *all* processors because the read is only globally performed when the reflected update has been globally performed. However, the processor can utilize the word and continue until a shared writable memory access becomes necessary. A processor that reads a word which is "old" for which *some processors* already have accessible a newer version may, however, proceed because the write associated with the returned "old' version of the word has been globally performed.

A system which meets condition 2.2 will generate the correct interleavings for the program fragments of Figure 2.16. Condition 2.2 can be applied to systems with fully replicated memories or to systems with only partially replicated memories, such as cache–based systems. Timing is of no consequence if condition 2.2 is met—the condition itself implies the necessary *order* without using a time reference.

2.9.2 Concurrent consistency

A concurrently consistent multiprocessor must be capable of correctly executing programs which require sequential consistency. Such a system requires *strictly sequencing processors* and a *word ordered memory* system.

Condition 2.3 (Concurrent Consistency) *A multiprocessor is concurrently consistent if all writes to the same word are performed with respect to every processor in the same order and if all writes performed with respect to a processor i before issuing a write s are also performed with respect to a processor j before access s is performed with respect to processor j. All accesses by the same processor must also be performed in program order with respect to all processors.*

The first part of condition 2.3 ensures general coherence, the second part ensures causal correctness, and the third part guarantees compatibility with sequential consistency. Note, that the third part of condition 2.3 dictates the order in which accesses by the same processor must be observed throughout the system. In a concurrently consistent system a processor can read a memory location and need not be aware whether the associated write has already propagated to all copies of the word. However, a processor must be assured its *previous* shared writable memory access has been performed with respect to the processor which caused the update of the word which the processor is now reading. For example, in the case of the simple synchronization protocol of Figure 2.8, process 1 (2) cannot read the test variable B (A) until the modification of A (B) has been performed with respect to process 2 (1). Otherwise the read would be performed

with respect to the other processor before the write has been performed with respect to that processor.

The easiest way to assure that all of a processor's accesses are performed in program order with respect to all processors is to disallow the issuance of a shared writable memory access until the previous shared writable memory access has been performed with respect to all processors. (A load does not necessarily have to be globally performed.) The first and second parts of condition 2.3—ensuring general coherence and causal correctness—must of course still be guaranteed by some other means.

It is clear that the ordering rule of condition 2.3 guarantees general coherence and the update rule satisfies causal correctness. Hence, condition 2.3 does indeed describe the necessary behavior of a system with strictly sequencing processors and a word ordered memory system. Furthermore, using the definition of concurrent consistency (2.6) it is apparent that by being generally coherent, causally correct, and strictly sequenced a multiprocessor is at least concurrently consistent.

While the condition for concurrent consistency is theoretically more relaxed than the one for sequential consistency it is much more difficult to enforce in most cases.

2.9.3 Weakly ordered systems

To maintain sequential or concurrent consistency, potential dependencies on *every* data access to shared memory have to be assumed. However, most of these data are not *synchronizing variables,* i.e., shared variables used to control the concurrency between several processes (such as variables A and B in Figure 2.8). In multitasked programs such variables are used to synchronize processes and to maintain the integrity of shared modifiable data structures or variables.

In sequentially and concurrently consistent systems most jockeying in queues is not allowed and often pipelining of accesses is difficult if permitted

because accesses have to be (globally) performed in order. Especially, in systems with complex packet-switched networks the network delay may be high, but a new access may not be issued until an acknowledgment has been received that the previous access has been globally performed. In most cases, though, performing accesses out of their order will not cause logical problems. In weakly ordered systems these restrictions do not exist for most memory accesses.

In a system with weak ordering of accesses, two types of shared variables are distinguished: first the shared operands appearing in algorithms whose values do not control the concurrent execution; and second synchronizing variables which protect the access to shared writable operands or implement synchronization among different processes. If a shared variable is modified by one process and appears in other processes and, if the access to the variable must be protected, then it is the responsibility of the programmer to ensure mutual exclusion for each access to the variable by using high-level language constructs such as critical sections [3]. Critical sections are in turn implemented by basic synchronization primitives such as locks. It is assumed that, at run time, the system can distinguish between accesses to synchronizing variables and to other shared variables. Synchronizing variables can be distinguished by the type of instruction (Test_and_set, Compare_and_swap, Fetch_and_execute, Reset or special load and store instructions, for example).

Condition 2.4 (Weakly ordered systems) *In a multiprocessor system, memory accesses are weakly ordered if (1) memory updates propagate to all copies of the addressed word in a finite amount of time, (2) accesses to global synchronizing variables are strictly sequenced and synchronization memory is either strictly coherent, write ordered, or word ordered, (3) consecutive accesses to non-synchronization variables use hazard-free sequencing, (4) no access to a synchronizing variable is issued by a processor before all previous global data accesses have been globally performed and if (5) no access to global data is issued by a processor before a previous access to a synchronizing variable has been globally performed.*

The dependency conditions on shared variables in such a system are not checked continuously but only at explicit synchronization points. Between two consecutive operations on hardware-recognized synchronization variables, no assumption can be made by the programmer of a process about the order in which stores are observed by other processes. The order of successive stores by a processor, to the *same* address, must however be respected. Buffering and restricted jockeying are allowed in all buffers, except for operations on hardware-recognized synchronizing variables.

In order that the program of Figure 2.8 executes correctly in a system with a weak ordering of memory accesses, variables A and B must have been declared as synchronizing variables. Special load and store instructions may therefore be generated by the compiler for such variables.

2.10 Access Ordering in Different Architectures

The rules to enforce sequential and concurrent consistency in multiprocessors have been defined in conditions 2.2 and 2.3, respectively. We now examine how these rules may be enforced in different multiprocessor system architectures.

We consider three different types of systems in the following discussion. These structures are representative of the system structures for multiprocessors with global data. In the following, the adjectives *shared* or *private* refer to the physical location of the memory. The adjectives *global* or *local* refer to the accessibility of the data. Global data are accessible by all the processors while local data are only accessible by the local processor. Similarly, a local buffer can only be accessed by one processor.

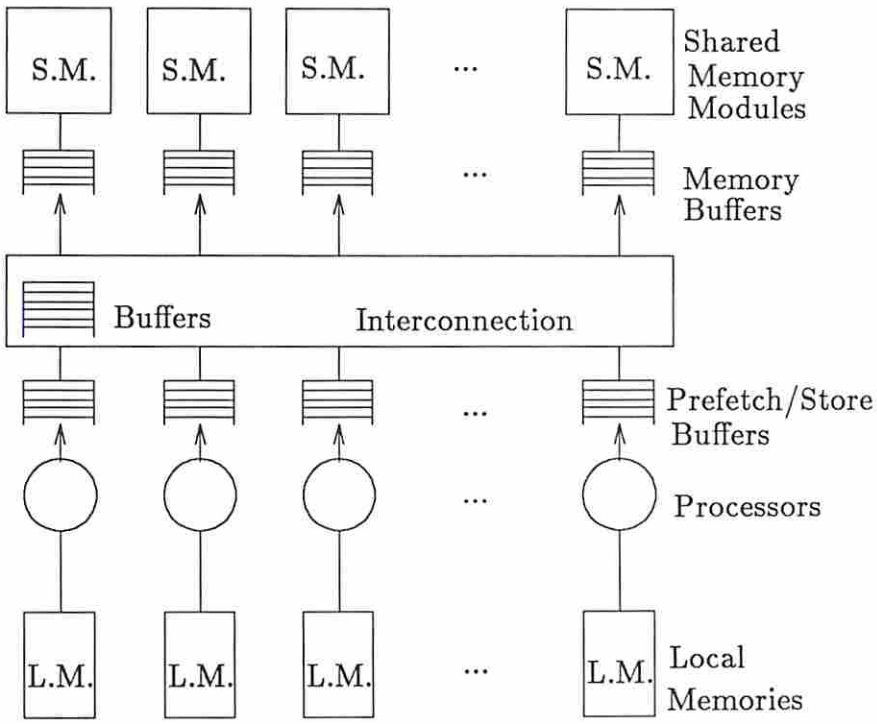


Figure 2.19: *System with shared global memory.*

2.10.1 Multiprocessor systems with shared global memory

The shared memory is interleaved and is accessed by all processors (Figure 2.19). Only one physical copy exists of each shared memory word. The memory is *strictly coherent* since any access is performed at the same time with respect to all processors as soon as it is buffered at the memory buffer. An example of this type of system is the C.mmp [33]. This system has been analyzed by Lamport [42]. It is assumed that the interconnection network is passive. If the delay through the network is constant or bounded for all requests (a rare case in practice because of conflicts), or if there is only one path from processors to memories (e.g., in single bus systems) then successive requests can be issued in program order without waiting for acknowledgments from the memory. In general, however, because of conflicts and random delays, the only way that a

processor can ensure that its global data requests are performed in program order is to issue the requests one at a time and to wait for an acknowledgment after each request. Once a request is buffered at the memory, it can be considered performed. The following are the rules for enforcing sequential consistency in the system shown in Figure 2.19.

1. Global memory accesses are performed once latched at the FIFO buffer of the destination memory.

2. Individual processors initiate global data accesses in program order. These accesses (both for loads and stores) are buffered in the same local buffer associated with the processor. The combined buffer is managed by a strict FIFO policy. Internal forwarding [36] (i.e., bypassing the memory) in a processor is restricted and governed by the rules of combining networks (see Section 2.10.4).

3. The controller of the combined Prefetch/Store buffer issues memory accesses one-by-one, in the FIFO order of the buffer. An access is issued once the previous one has been latched at the destination memory buffer or when it becomes impossible for the to-be-issued access to be latched at its destination memory buffer *before* the previous access has been latched at its destination memory buffer.

The essence of ensuring sequential consistency in multiprocessors with strictly coherent memories is to guarantee that accesses by the same processor are latched at the memory in program order.

2.10.2 Multiprocessor system with distributed global memory

A multiprocessor with distributed global memory which consists of an interconnection of private memories is considered here. Several existing multiprocessors fall into this category, such as the Cm*, the BBN Butterfly, and the

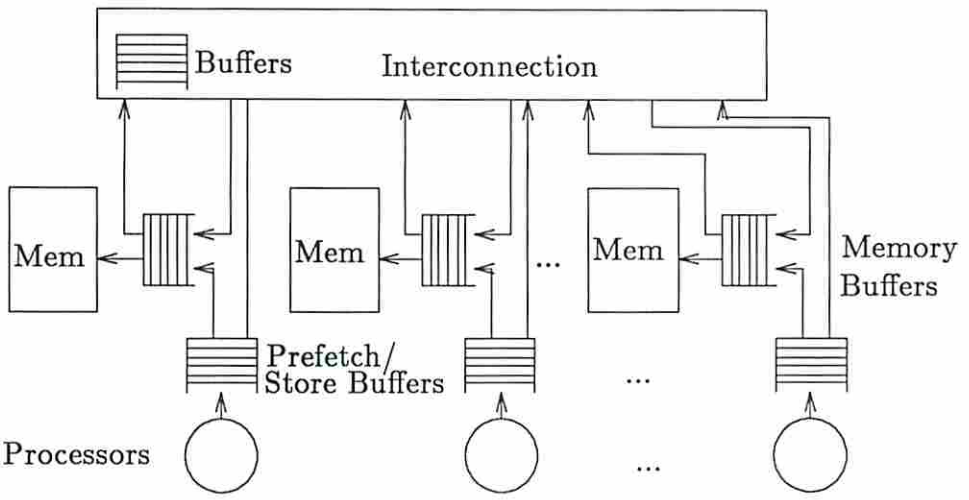


Figure 2.20: *System with distributed global memory. Remote accesses are buffered in Memory buffers.*

IBM RP3, [33,16,50]. The private memories contain global, as well as local data and are accessed randomly (no multiple copies of data exist) (Figures 2.20 and 2.21). Memory is strictly coherent.

The systems depicted in Figures 2.20 and 2.21 differ in that memories in the second system are accessed via two separate buffers, while the system of Figure 2.20 uses only one buffer per memory. In the system of Figure 2.21, the Prefetch/Store buffer queues accesses from one processor only (the processor which *owns* that particular memory module), while the Remote Access buffer queues accesses coming from all other processors. In the system of Figure 2.20 all accesses to a particular memory module are queued in the same Memory buffer.

It should be evident that the system depicted in Figure 2.20 is logically equivalent to the system shown in Figure 2.19. Access delays due to remote and local accesses are different in system 2.20, but such delays were never taken into consideration when the conditions for sequential consistency were laid down in the previous subsection. Hence, the conditions for sequential consistency in system 2.20 are the same as those for the system shown in Figure 2.19. Any

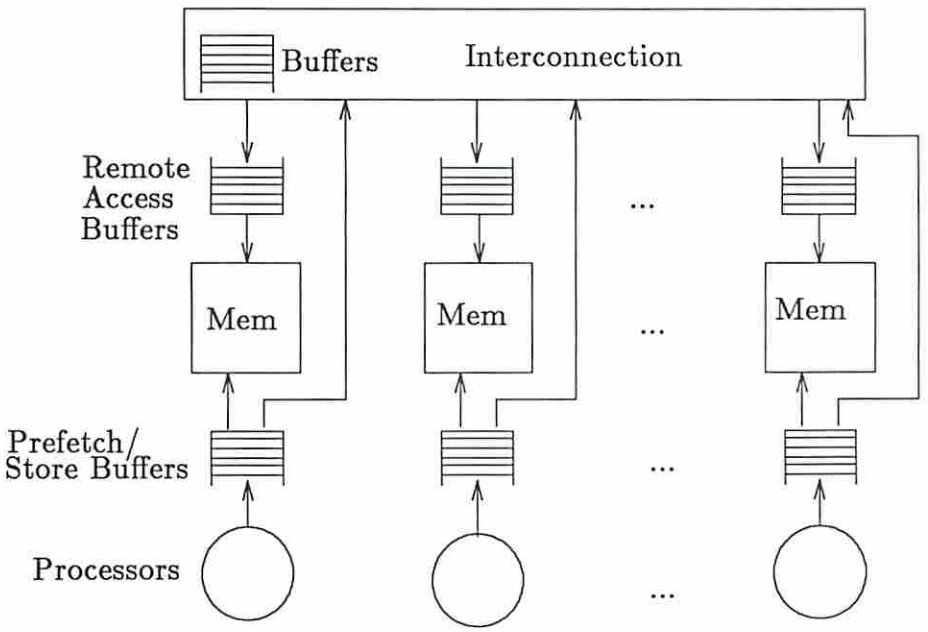


Figure 2.21: *System with distributed global memory. Remote accesses are buffered in Remote Access buffers.*

access is performed when it is buffered in the Memory buffer of the destination memory. All accesses are *initiated* by being placed in the Prefetch/Store buffer of the processor. A private access is *performed* as soon as it is *issued* by the Prefetch/Store buffer to the local Memory buffer. A remote access is *issued* by the Prefetch/Store buffer by introducing it to the interconnection. The access is *performed* when it is latched in the destination Memory buffer. It should be evident that a Prefetch/Store buffer should wait until the most recent remote access has been performed before it can issue another access. A well-known example of the system shown in Figure 2.20 is the Cm* [33].

In the system depicted in Figure 2.21, remote accesses are buffered in a distinct remote access buffer. This system, in its general form, is not strictly coherent. Once an access is buffered in one of the two queues it is only performed with respect to the processor(s) which use(s) the same queue (provided that queues are accessed in FIFO order). The policy used in sequencing accesses queued in different buffers will affect the logical properties of the system.

If accesses by the two buffers are interleaved randomly, then no access is globally performed until it is executed at the memory. The Prefetch/Store buffer, however, preserves the order of the accesses initiated by the local processor. Remote accesses initiated by the local processor are only performed when they are executed in the remote memory. This strategy conforms to condition 2.2. This scheme may be inefficient, because in the case of a remote access the Prefetch/Store buffer is blocked until the remote access has been dequeued at the appropriate Remote Access buffer. If Remote Access buffers often contain many pending accesses, this blocking time can be relatively long.

One way of avoiding the blocking of the Prefetch/Store buffer is to assign an absolute priority to Remote Access buffers. This means that all accesses waiting in Remote Access buffers are serviced before accesses waiting in Prefetch/Store buffers. When a remote access is buffered at the Remote Access buffer it can be considered globally performed and the local access buffer which issued it may proceed without blocking. A remote access is globally performed when it is buffered at the Remote Access buffer of memory module X because:

1. It is performed with respect to all processors which access memory module X via the Remote Access buffer (this is due to the FIFO nature of this buffer).

2. The access is performed with respect to the processor which accesses memory module X via the Prefetch/Store buffer, because of the absolute priority of the Remote Access buffer over the Prefetch/Store buffer.

The time to perform an access in system 2.21 may be quite long, if the buffers are not prioritized. Weak ordering may be more efficient, as supported by the simple model of Section 2.11. In the case of weak ordering, the Prefetch/Store buffer issues local references by starting the memory cycle and issues remote references by simply latching them in the first stage of the interconnection. However, whenever a processor executes an instruction on a declared synchronizing variable (this is detected by the fact that the data have been tagged by the compiler,

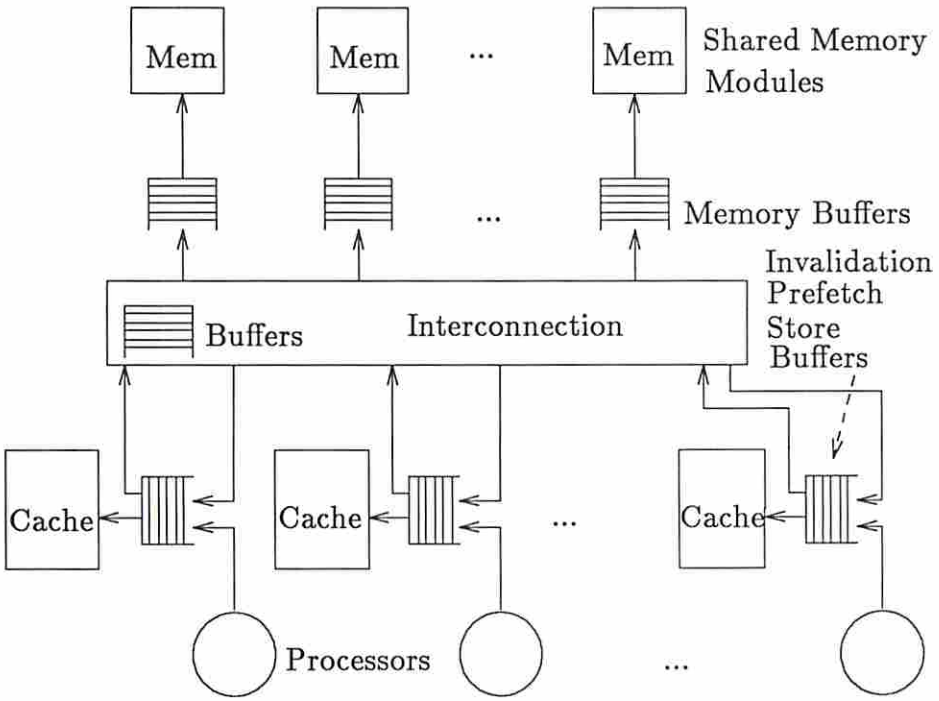


Figure 2.22: *Cache-based system. Invalidations are buffered in the Prefetch/Store buffers.*

or that special instructions are used), it must ensure that all its previous data accesses have been globally performed, and stop issuing prefetches and stores of operands until the access to the synchronizing variable is also globally performed.

2.10.3 Multiprocessor systems with private caches

Figures 2.22 and 2.23 show multiprocessor architectures with shared global memory and private caches. One example of this type of system is the Sequent Symmetry multiprocessor [56]. The shared memory contains code and data, and the caches are accessed associatively.

Caches may be write-through or write-back caches [59]. If no global writable data can be loaded into caches then no coherence problem exists between the caches. This technique relies on software to avoid the coherence problem. At any time, the caches contain local data or non-modifiable global data. The

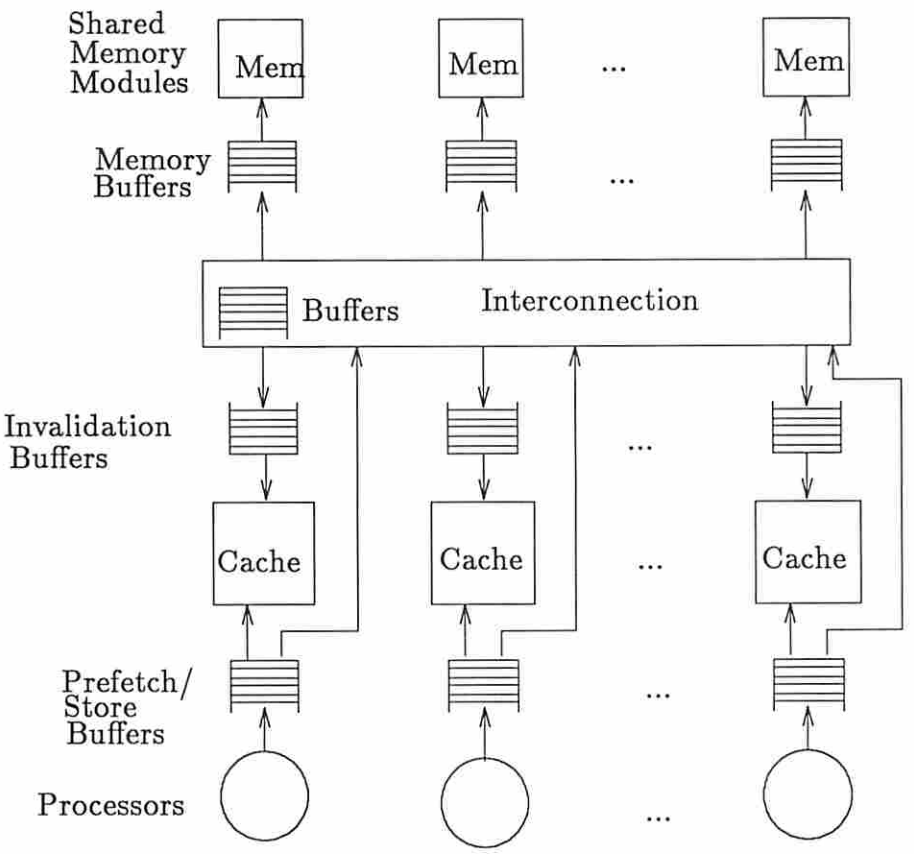


Figure 2.23: *Cache-based system. Invalidations are buffered in independent Invalidation buffers.*

distinction between the types of data is made at compile-time, possibly with some indication from the programmer. Accesses to global writable data in the shared memory can be buffered at the processor in a common Prefetch/Store buffer. With respect to buffering, the problems with cache-based systems in which the cache never contains shared writable data are very similar to the problems analyzed in Section 2.10.1.

Of particular interest is the case of multiprocessors with private caches that can contain global data and with hardware-enforced coherence [12,18]. Coherence among multiple copies of these data can be maintained through hardware invalidation signals. Also, in some coherence algorithms, a processor may broadcast a load to all caches in the case of a miss, in order to read the data directly from another cache. In the following, "P-data" refers to data that are private to the cache (one single copy exists) and "S-data" refers to data that are shared among several caches (several copies may exist in different caches) [18].

Two buffer configurations are shown in Figures 2.22 and 2.23. In Figure 2.22, there is a unique buffer per processor. The buffer contains data prefetch and store requests for the local processor plus the accesses made by remote processors (loads or invalidations). The local processor initiates its stores and loads in the private Prefetch/Store buffer. No jockeying is allowed in this buffer. The private buffer controller issues requests to the cache one at a time. A store request on S-data in the cache may require invalidating the data in other caches. A store issued (Figure 2.22) by processor i is performed with respect to processor k when the cache is updated (store hit on P-data) or at a point in time when it is placed in the memory buffer and when an invalidation (if it is necessary) has been placed in the local buffer of processor k (store to S-data). Similarly, on a read miss, a cache may read a block from a different cache. A load request on a miss in processor i is performed with respect to processor k when the load request is buffered in the local buffer of processor k.

The loads causing misses and the stores on S-data causing invalidations can be broadcast on a bus to all caches and to memory so that they are performed

with respect to all processors at the same time. This means that loads and stores can be performed atomically; this solution can be extended to systems with a few buses.

In the system of Figure 2.23, the buffer for invalidations and for loads issued by remote processors is distinct from the buffer for loads and stores from the local processor. An invalidation of a block in a remote cache of processor k is performed with respect to processor k when it is executed in the remote cache. It is difficult to pinpoint the instant when a store is performed with respect to a processor because the invalidation buffer of each cache may contain different numbers of invalidation requests, making the time to invalidate each cache unpredictable.

In a weakly ordered system, the processors can issue shared memory requests without waiting for previous requests to be performed. This results in a system with very high efficiency. In this case, the only troublesome accesses are accesses to synchronizing variables. The buffer controller must still record the status of all cache accesses that it has issued but not performed, so that it can perform them every time an access to a synchronizing variable is detected. The implementation of such a buffer may be very complex. The possibility of deadlocks must also be considered. The details of an implementation for a given cache coherence mechanism would be interesting in order to understand the practical aspects of the concept of weak ordering in cache-based systems. In Chapter four some aspects of weakly ordered, cache-based systems are discussed. In particular, how a cache-based multiprocessor can be lockup-free and maintain weak ordering of accesses is discussed in Chapter four of this thesis.

Consider again the system of Figure 2.22. But this time, let us assume that invalidations are not broadcast on a bus but rather are propagated from cache to cache by the interconnection. It is interesting to note, that sequential consistency is preserved in this system, provided the memory copy is locked during the propagation of the invalidations and provided that the invalidated caches cannot obtain a "new" copy of the invalidated block until all other caches

have been invalidated as well. The details on maintaining sequential consistency in such a system are discussed in the next Chapter.

2.10.4 Multiprocessor systems with combining networks

In the previous analysis, we have assumed that the network does not combine accesses, i.e., no access can be performed with respect to *any* processor while it is in transit in the interconnection network. Combining networks such as the one proposed for the NYU ultracomputer [31] are not passive. In this Section we analyze the logical properties of such networks for sequences of loads and stores in the light of condition 2.2.

2.10.4.1 Combining of ordinary load and store accesses

When memory accesses "collide" in a switch box of a combining network the following combinings occur, depending on the type of access.

1. Case of multiple loads to the same variable x: only one load is propagated to memory. The other loads are buffered in the switch node until the return of the value from the memory; then the value is returned to all processors having issued the load.

2. Case of multiple stores on the same variable x: Only one value defined by one of the stores is propagated to memory. Acknowledgments are returned to all processors having issued the stores.

3. Case of multiple loads and stores on the same variable x: Only one value defined by one of the stores is propagated to memory. Acknowledgments or data values defined by the store are returned to all processors having issued the loads or stores.

Memory accesses can terminate in three ways. Either an access terminates at the memory, or it terminates at a switch with combining, or it is eliminated at a switch. In case (1), all but one load terminate at the switch

where the combining takes place. These loads wait for a value to be returned by the not yet terminated load which propagates towards memory. This load either combines again with other accesses at another switch or may terminate at the memory. A load which terminates at the memory is globally performed when the store which defined the value seen by the load has been globally performed (Definition 2.9). As soon as a load terminating at the memory is globally performed, all loads which are waiting for a value to be returned by the load (since they combined with it) are also globally performed. If loads waiting at a switch for a value to be returned by the load propagating towards memory have the possibility to combine with a store to the same memory location, then they may do this under the provisions described for combining loads and stores (case 2).

In case (2) the load(s) combining with a store terminate at the switch. Such loads are not globally performed until the store which they combine with is globally performed. The loads may return the bound value to their respective processors, but assuming condition 2.2, the processors may not issue another global memory access until the store propagating towards memory has been globally performed. A store terminating at memory is globally performed once all copies of the to be written data have been updated. A store which has previously combined with loads may also subsequently combine with more loads or other stores (case 3).

In case (3) all but one store are eliminated at the switch. A single store propagates towards memory. The stores combining may have previously combined with other stores or loads. A store which is eliminated by combining with other stores is globally performed once the remaining store, propagating towards memory, has been globally performed. This is due to the strict interpretation of definition 2.9 which states that an access is not globally performed if a subsequent load can return an "old" value. Even though an eliminated store may have no impact on the system, if sequential consistency is to be preserved, then a "new" value must become visible due to the eliminated store. A store may be eliminated even if it has previously combined with loads.

Note, that the above rules refer to correctness under sequential consistency. Implementing these rules may be difficult. For example, if loads combine with a store and wait for some type of acknowledgment for the store to be globally performed, the implementation of the acknowledgment mechanism will be complicated if the store can be eliminated in transit to memory.

2.10.4.2 Combining of atomic Read-Modify-Write accesses

The Fetch_and_add instruction returns the value of a memory location and increments it atomically by a selectable value. It is desirable to allow multiple Fetch_and_Adds to combine in an interconnection, even if ordinary loads and stores cannot combine. This is due to the fact that the destination of Fetch_and_add accesses often forms a *hot spot* (i.e., a memory location for which contention is high). Multiple Fetch_and_adds can combine in a switch by buffering all but one Fetch_and_add access at the switch and forwarding a single modified Fetch_and_add which returns the unmodified memory location value and increments it by the sum of the increment values of the combined Fetch_and_adds. Multiple modified Fetch_and_add operations may also combine in later stages of the network. Each buffered Fetch_and_add instruction returns a differently offset value of the value returned by the Fetch_and_add which terminated at the memory. Fetch_and_add instructions are globally performed once an ultimate modified Fetch_and_add terminates at the memory.

Other indivisible read_modify_write operations can combine at switch nodes under similar rules. Likewise read_modify_write operations may also combine with ordinary load or store accesses to the same memory location.

2.11 Simple Performance Analysis

In this section we present a simple model comparing the upper bounds of the MIPS rates achievable by three designs: a design with no buffers at the processors, a design using strict sequencing and restricted ordering to maintain

sequential or concurrent consistency as specified by conditions 2.2 and 2.3, and a weakly ordered design. Each processor alternates between compute and wait phases. A processor "computes" while it accesses local data and instructions. On a reference to global data, it has to wait for the completion of the access. This waiting time depends on the type of the access.

Let t_p be the average duration of the compute phase between two successive data accesses to global memory in one of the P processors. If p_s is the probability that an instruction contains an access to global data and $I_{s.p.}$ is the MIPS rate of a processor when all accesses are local (single processor configuration), then $t_p = 1/(I_{s.p.}p_s)$. The memory system is characterized by t_{issue} and $t_{perform}$, the minimum times to issue and to perform a request, respectively. All these parameters depend on the machine design.

To define the upper bound of the MIPS rate, we neglect memory access conflicts as well as dependencies in the CPU. We also assume that all buffers are infinite. The following results show the relative effects of the three design approaches. Let t_{inref} be the average interreference time between two consecutive accesses to global variables by the same processor, i.e., it is the total duration of the compute and the wait phases between two accesses to global memory; the following inequalities define upper bounds of the MIPS rate of the three systems.
$t_{inref} \geq t_p + t_{perform}$ (no buffering at the processor),
$t_{inref} \geq MAX[t_p, t_{perform}]$ (buffering with strict sequencing), or
$t_{inref} \geq MAX[t_p, t_{issue}]$ (buffering with weak ordering).

The first inequality results from the fact that, in a non-buffered system, the processor is blocked during the time it performs globally a shared memory access. In the second or third cases, the processor and the Prefetch/Store buffer controller form a pipeline with average segment times of t_p and $t_{perform}$ (strict sequencing), or t_p and t_{issue}, (weak ordering). The throughput of this pipeline is determined by the bottleneck segment.

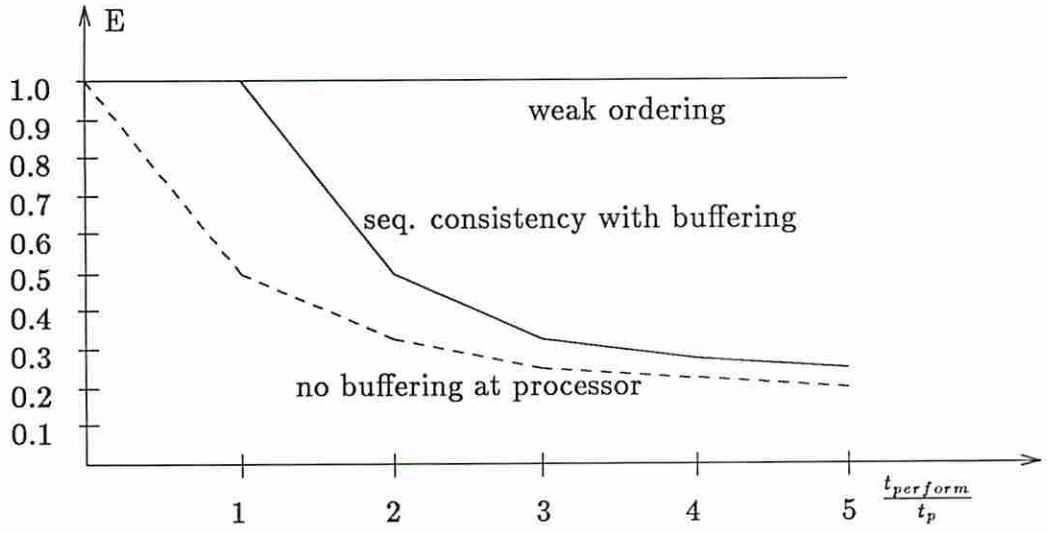


Figure 2.24: *Comparison of systems without buffering (dashed line) and with buffering and sequential consistency (solid line). Comparison of buffering with sequential consistency and weak ordering* ($t_{issue} < t_p$).

The efficiency of the multiprocessor system denoted E is the ratio of the MIPS rates of a processor in the tightly-coupled ($I_{t.c.}$) and in the single processor ($I_{s.p.}$) configurations.

$$E = (I_{t.c.}/I_{s.p.}) = (t_p/t_{inref}).$$

From the above formulas, we can see that the effectiveness of various designs depends on the relative values of t_p and t_{issue} or of t_p and $t_{perform}$. Note that the value of t_p depends both on the MIPS rate of the processor as a single processor, and on the probability that an instruction references data in global memory. In Figure 2.24 the upper bounds on multiprocessor efficiencies in the cases of no buffering at the processor and of buffering with strict sequencing are compared as a function of $t_{perform}/t_p$.

Buffering is more attractive for highly-pipelined machines, and for cases where the access rate to global memory is high. For the third curve of the curves in Figure 2.24, we have assumed that $t_{issue} < t_p$ and the lower bounds on multiprocessor efficiency of buffered sequentially consistent systems is shown as

$t_{perform}/t_p$ increases. This would be the case if the number of processors increases and the interconnection network is packet-switched so that the delay through the network increases, but the time to issue remains the same. It appears from the lower bound model, that weak ordering in buffered system is only effective for systems where $t_p < t_{perform}$.

2.12 Conclusion

We have presented in this Chapter a framework to analyze the coherence properties of shared memory multiprocessor systems. The concepts and results presented in this chapter are extensions of Lamport's results [41,42,43] and Collier's work [14]. By using the notion of *event ordering* several key concepts of shared memory multiprocessors have been introduced.

Two types of processor memory-access sequencing methods have been found to be relevant.

1. Strict sequencing allows a processor only to issue another memory access after the previous accesses have been "completed." Strict sequencing is necessary to preserve *all* potential dependencies among different instruction streams.

2. Hazard-free sequencing is commonly used in uniprocessors and allows a processor to issue memory accesses in any order which does not potentially violate *intra-process* dependencies.

Only strict sequencing enables a system to conform to concurrency models which allow control of execution sequencing and synchronization by using ordinary shared writable memory.

Restricting the individual processors' behaviors is not sufficient to ensure any desired system behavior (concurrency model). The memory system as a whole, that is, including interconnections, must also conform to a particular behavior model. Several memory properties were derived and defined.

1. Strict coherence refers to the property of a memory system which serializes all accesses to any given word by all processors. Strict coherence conforms to what has traditionally been referred to as "memory coherent."

2. A write ordered memory was defined by Collier and refers to a memory system in which all replicated copies of memory experience the exact same order of word updates.

3. A memory system is generally coherent if all copies of a word will always reflect the same value after some finite time of no write accesses to the word. In a generally coherent memory system each copy of a word is "logically" updated in the exact same order.

4. A memory system is causally correct if via the memory system an "event" cannot be observed before the "cause" of the event. That is, the "relaying" of information in a causally correct memory system is restricted.

5. A memory system is word ordered if it is both causally correct and generally coherent.

6. A memory system is non-ordered if it is neither strictly coherent, causally correct, or generally coherent. Such a memory system must generally be supported by other synchronization hardware which may be, for example, a smaller strictly coherent memory.

Three useful concurrency models result from combining processor and memory models.

1. A sequentially consistent multiprocessor can use ordinary shared memory accesses to synchronize processors and communicate among processors. By means of software alone it is impossible to determine whether a program was executed on a sequentially consistent multiprocessor or a multitasking uniprocessor. Sequential consistency is a restrictive concurrency model

and guarantees compatibility with code written for most multiprocessors—independent of architecture. A sequentially consistent system requires at least a write ordered memory system and strictly sequencing processors.

2. A concurrently consistent multiprocessor models a concurrent system which preserves causality. It can correctly execute most code intended for sequentially consistent multiprocessors. It is, however, possible to determine by software alone that the system is not sequentially consistent. The code required for this determination serves no purpose in a sequentially consistent multiprocessor. A concurrently consistent multiprocessor requires strictly sequencing processors and a generally coherent and causally correct memory system.

3. A weakly ordered multiprocessor utilizes a non-ordered memory system. A weakly ordered system cannot implement synchronization and reliable inter-process communication without special hardware. However, a weakly ordered system allows the less restrictive hazard-free sequencing of accesses by processors.

To utilize the conditions under which a system conforms to a desired concurrency model three memory access states were defined. These states, (*initiated, issued,* and *performed*) correspond to memory accesses states of

- out of processor control but not yet global
- global, in the interconnection, but not yet completed, and
- completed, at least with respect to (some) remote processor(s).

These concepts have been analyzed briefly for some specific systems. Contrary to the "common wisdom" a processor in a multiprocessor system does not have to block on every shared memory accesses for the system to be sequentially consistent or otherwise "safe." Multiple global memory accesses may be in progress at any time, provided that conditions 2.1, 2.2, 2.3, or 2.4 are respected.

These conditions permit a considerable flexibility in the buffering of accesses at different levels of the memory system. More astonishing is probably the fact that strict coherence is not a necessary condition for sequential consistency in asynchronous multiprocessors.

A simple model has been presented to give insight into the effectiveness of various design approaches for different ranges of system parameters. The fundamental approach taken in this Chapter has allowed us to identify the basic restrictions on buffering imposed by the three concurrency models in the case of some very complex systems, such as cache-based systems.

Chapter 3

SEQUENTIAL CONSISTENCY AND CACHE–BASED MULTIPROCESSORS

In this Chapter the applicability of concurrency models to the design of multiprocessors is demonstrated. The enforcement of sequential consistency in systems with private caches is analyzed. It is shown under what conditions a cache–based multiprocessor is sequentially consistent and hence can easily implement synchronization and communication. Since cache–based systems pose special problems a new set of definitions is introduced to make directly applicable the conditions of concurrency models defined in the previous Chapter. It is also shown that cache–based systems can indeed implement indivisible read-modify-write memory accesses to simplify synchronization.

The advantage of examining cache–based multiprocessors in the context of sequential consistency versus the traditional approach of proving systems memory coherent is that memory coherence assumes a global word access order which can be difficult to "impose" on the system. By proving a multiprocessor sequentially consistent, it follows that the system also behaves as if it was memory coherent (*strictly coherent*). Furthermore, the conditions for sequential consis-

tency are much more easily applied to existing protocols, providing unambiguous guidelines for correctness.

3.1 Introduction

The performance of shared-memory multiprocessors is limited by memory access conflicts. A usual means of reducing these conflicts is to associate a private cache with each processor [12,51,59]. In a multiprocessor system, the use of private caches poses the familiar and complex problem of cache coherence: The problem of how to maintain data coherence among the various copies of data which can reside in multiple caches and main memory, in a manner that is completely transparent to the user of the machine. This latter condition is highly desirable in a general-purpose environment. Many verified solutions to the problem of multiprocessor cache coherence have been proposed and several have been successfully implemented. Usually hardware and software cooperate to implement *strict coherence*. Much effort has been devoted to the minimization of the overhead traffic necessary to maintain coherence on single bus systems [5,30,48]. As a result of minimizing overall traffic, the complexity of the coherence protocols has increased. To prove most of these efficient protocols correct has become exceedingly difficult.

Cache coherence protocols should provide for the execution of indivisible read-modify-write instructions, such as the Test&Set instruction. The implementation of indivisible (atomic) instruction sequences greatly simplifies inter-processor synchronization. To be able to execute parallel programs written for multitasking uniprocessors, cache–based, shared memory multiprocessors should also be sequentially or concurrently consistent. As it is difficult to enforce concurrent consistency in most cache–based multiprocessors (the enforcement of general coherence is implementation specific), this Chapter will focus on the enforcement of sequential consistency in such machines. Sequential consistency conditions can be applied to meet the following goals of a system designer:

- Demonstrate that a cache coherence protocol is correct by showing that it results in a sequentially consistent multiprocessor.

- Design new protocols for cache-based multiprocessors, which *do not* require a single-access broadcast interconnection to propagate data updates and invalidations.

- Implement the correct execution of indivisible instruction sequences, such as the Test&Set instruction, in any system that uses an *ownership* protocol.

In traditional cache coherence schemes there is a single-access broadcast system, usually a bus. Traffic through the bus is a direct result of the communication and sharing of data between processors. It is unlikely that this traffic can be reduced indefinitely through more "clever" coherence protocols, mostly because in a general–purpose environment, tasks are diverse and the sharing of data between them can take different forms; more importantly, there are intrinsic lower bounds on the amount of sharing required by parallel algorithms to solve certain problems.

In this Chapter we avoid the notion of *memory coherence* to analyze cache-based multiprocessors, but rather use the concept of *sequential consistency.* The reason for our approach lies in the fact that the traditional definition of memory coherence fails to provide sufficient indication on how to enforce it in systems where writes are not atomic. In systems where writes are not atomic, updates may become *available* at different times to different processors. To accomplish atomic updates, either single buses are used to perform all global accesses or a dedicated invalidation bus is employed to broadcast invalidations only. The requirement for at least one bus to which *all* processors must be connected inherently limits the possible number of processors in the system. This limitation is a result of the electrical problems posed by connecting many devices to a single bus, as well as of the increase of contention on the bus when the bus becomes saturated with requests.

In the previous Chapter, we have described how generic multiprocessor systems can operate correctly depending on whether the user expects the system to behave in a *sequentially or concurrently consistent* or in a *weakly ordered* fashion. In a *weakly ordered* system, coherence is only enforced at synchronization points. Between two successive synchronization points, a programmer does not make any assumption about the order in which updates are observed by processors.[1] Bitar and Despain [11] have shown how weak ordering can be easily upheld by implementing synchronization primitives (called *hard atoms*) correctly in cache-based systems with a global broadcast capability.

Sequential consistency is the strongest requirement possible on the ordering of events in a multiprocessor because it imposes the condition that processors may not observe stores in different orders. Systems that behave in a sequentially consistent manner are very conventional. They include the C.mmp [33], the Synapse System N+1 [25], and the proposed Spur system [32]. All parallel programming languages we know of present the programmer with a model of execution which either adheres to sequential consistency [3] or a model of execution which is not violated if the system is sequentially consistent. While synchronized algorithms rely on explicit synchronization for correctness, there exists also a class of algorithms, called *asynchronous algorithms* in [38] and [9], in which all synchronizations have been removed and which work because processes can read and update shared variables without synchronizing.

Several papers in the literature have applied the sequential consistency condition to show the correctness of multiprocessor behavior. Lamport has applied similar reasoning to distributed, message-based systems [41] and memory accesses in shared memory systems with no cache [42]. Rudolph and Segall proved the correctness of a *snoopy cache* protocol on a single bus. In [6] Baer provides a model, based on Petri nets, to prove general cache coherence protocols correct. Because the model is very intricate, it may be difficult to guarantee its correct

[1] The only condition on ordering of memory accesses is that dependencies within each individual instruction stream must be enforced.

application to a complex cache coherence protocol. The model does not presume sequential consistency, but we believe that the stated conditions for coherence would also result in a sequentially consistent system. Collier [14] has developed a theoretical framework based on possible execution graphs to study the coherence properties and ordering of events in a multiprocessor where each processor owns a replica of the whole shared memory. Approaches similar to Collier's have been used to analyze the correct sharing of registers by asynchronous hardware [65].

Using conditions 2.1 and 2.2 of the previous Chapter the following condition for sequential consistency in cache–based systems can be derived.

Condition 3.1 (Sequential consistency in cache–based systems) *Sequential consistency is satisfied in cache–based systems if an access may not be performed with respect to any processor until the previous access by the same processor has been globally performed and if accesses of each individual processor are globally performed in program order.*

This condition is powerful because it is easily applicable in practical systems and provides a solid basis for proving non-trivial cache coherence protocols correct. Note that this condition is sufficient but not necessary. (In Section 3.5, it will be shown that the above condition can be violated in a multiprocessor with *isolated processor-cache* modules while retaining sequential consistency. It will be shown that the outcome of a program is *indistinguishable* from the possible outcome of a system strictly adhering to the above condition.) Based on condition 3.1 it can be stated that any system is sequentially consistent if the following are true:

A: All processors issue memory accesses in program order.

B: If a processor issues a store, then the processor may not issue another shared–memory access until the value written has become *accessible* to *all other processors.*

C: If a processor issues a load, then the processor may not issue another memory access until the value which is to be read has both been *bound* to the load operation and become accessible to all other processors.

The following is an informal proof of the above conditions: Multiprocessors such as the C.mmp are classified as sequentially consistent [42] because of the following reasons:

1. Processors execute statements in program order.
2. Memory accesses are performed one-by-one; an access i is not issued until access i-1 has been completed.
3. Stores are atomic: i.e., a stored value becomes readable at the same time by all processors.

We note that no notion of timing is present in the above conditions; that is, even if accesses are delayed by a random but finite amount of time, the system remains sequentially consistent. Point (3) is not a strong condition, if the system *somehow* maintains conventional memory coherence, such as provided by an invalidation bus scheme. Hence, a system governed by condition 3.1 behaves as dictated by the above three points for the following reasons:

I. Point (1) is maintained per definition of condition 3.1.

II. Point (2) is maintained per definition as well, since an access *cannot be more complete* than when it has been performed.

III. When condition 3.1 is upheld, the effect of reads on a variable is delayed until the store defining the value is performed; therefore, for all practical purposes, one can consider that the stores are performed atomically at the point in time when the value is readable by all processors.

The outcome of (III) depends on whether or not a write to the memory location to be read is presently in progress. A read x may not proceed until the

latest write x ← D has *settled.* This means that at any time a processor may read a word x, reflecting the n^{th} update of x, but any processor which at that time successfully reads x, reflecting the $(n+1)^{st}$ update, must not issue another shared–memory access until the $(n+1)^{st}$ update of x has become available to *all* processors.

3.2 Read Paths, Owners, and Keepers in Coherence Protocols

In the previous Chapter, the different states in which a memory access may be at any time were defined. In a system that maintains single copies of data, it is simple to analyze in what state an access is in at any time. However, in systems where multiple copies of data exist, the different states in which an access may be are not easily analyzed. For such systems, reads and writes often imply other activities such as invalidations or broadcasts of data. By redefining what exactly any read or write implies in cache-based systems and by classifying copies of data, it becomes simple to recognize when exactly any access is *performed* or *globally performed.*

3.2.1 Read paths

In a conventional cache-based system with invalidations [5], a write is performed with respect to a processor k when both the write has been queued at the main memory and the cache which belongs to processor k has been invalidated. If one of the two conditions has not been met, then stale data may be read by a processor. Hence, *performing a write with respect to a processor k* implies not only that a copy of the data must be updated, but also that processor k must be made aware of where to find the data. We refer to the pointer to valid data as a *read path* that leads from the cache to main memory. A write consists of setting read paths and modifying the data. In the normal invalidation approach, a read

path to main memory is *implied by invalidating the block.* In other schemes, the read path may implicitly point to a table that in turn explicitly points to a valid copy of data contained in some other cache.

A write is globally performed once the read paths of all caches have been modified (if necessary) to point to valid copies of data and once all copies of data to which read paths point have been updated to reflect the write operation. A read is globally performed when it is guaranteed to return a particular value and when the write which *caused* that value has been globally performed.

3.2.2 Owners and keepers

When a cache or main memory contains a valid block, then this block may be the only copy or one of several valid copies of the block. If several copies of the block exist, then the storage devices containing them are *keepers* of the block. If the cache or memory contains the only copy of the block, then the storage device is the *owner* of the block. Any owner is also a keeper. If a write is performed to a keeper of a block, then all other keepers must be updated. If a write is performed to an owner of a block, then no further action need be taken to maintain sequential consistency. Data may be read from any keeper or owner. If a read operation obtains the copy of the block from an owner and the block is allocated, the read must be handled specially since the owner will cease being an owner. If there is an owner, then there is always only a single owner. In a write-through system, main memory is always a keeper, and, if there is one at all, main memory is always the only possible owner. Hence, we distinguish among four different events which occur in cache-based multiprocessor systems, regardless of the cache coherence scheme used.

1. MODIFICATION OF STORAGE. This event occurs at either a cache or main memory. The storage device must either be a keeper or an owner, or will become one of the two. A single write, as perceived by the processor, may cause several memory modifications, such as a broadcast of writes if

several existing keepers need to be updated to maintain coherence. Often, a write consists not only of the modification of a word but also of the transfer of an entire block which consists of several words. This may be the case, for example, in a write-back system when a dirty block is replaced in a cache.

2. MODIFICATION OF A READ PATH. This modification makes known to the processor the location of the copy which is currently valid. Theoretically, the read path may consist of a finite number of "pointers" (i.e., some type of address to identify memories, caches, and tables) which the reading processor can follow to find valid copies of a block. If a block is present and valid in a cache, then the read path is simply "nil". Usually a read path implies either: a local copy of a block (path is nil); a non-present block which implies a table lookup (path to table) which in turn either points at a valid block in another cache (path to specific cache) or main memory (path to memory); or an invalidation which directly implies that the block must be fetched from main memory (path to memory). Care should be taken that a read path under modification can not be accessed or that an access will always yield either the "new" or the "old" version of the block. A single read path may only be modified by one write at a time. Separate read paths all leading to the same block *can* can be modified concurrently *if* eventually all read paths will point to the same data (i.e., the system remains generally coherent). (This serialization, however, does not mean write atomicity.)

3. FOLLOWING A READ PATH AND READING DATA. We normally do not distinguish between following a read path and actually reading. The two activities are the states in which a read may be, and they are analogous to a read which is issued and a read which is performed.

4. BECOMING AN OWNER AND RELEASING OWNERSHIP. Not all storage devices contain valid copies of all blocks at all times; that is, not all storage devices are keepers of all blocks. However, all storage devices main-

tain valid read paths to keepers, by default. This is the reason why a switch of keepership (i.e., becoming a keeper) is not included in our list of events. A switch of keepership manifests itself as a read path modification. For ownership changes, though, an event type is defined. Ownership changes are caused either by a processor that for some reason (usually a write) wishes to acquire ownership, or because ownership may simply be lost by a cache when it replaces an owned block. Ownership by caches exists only in write-back systems, because in write-through systems main memory is always a keeper. Since an owner must be the only keeper, no cache can ever be an owner in a write-through system.

The above events are triggered by loads, stores, and block replacements. One store may cause all four events; for example, the issuing processor's cache becomes the new owner (event 4), the block is fetched from a previous keeper (event 3), the block is written to the new owner (event 1), and all other caches must be "made aware" that the block can now be found at the new owner only (event 2).

3.3 Sequential Consistency in Bus-based Systems

We wish to establish the usefulness of conditions A, B, and C defined in Section 3.1 to demonstrate the correctness of cache coherence protocols.

3.3.1 Snoopy cache protocol

In [51] Rudolph and Segall propose a cache coherence protocol of moderate complexity. In the same paper the protocol is extended to accommodate an interleaved cache system which connects each processor to several private caches, with the result that communications may be spread over several different buses. In the following, both approaches are shown to be correct.

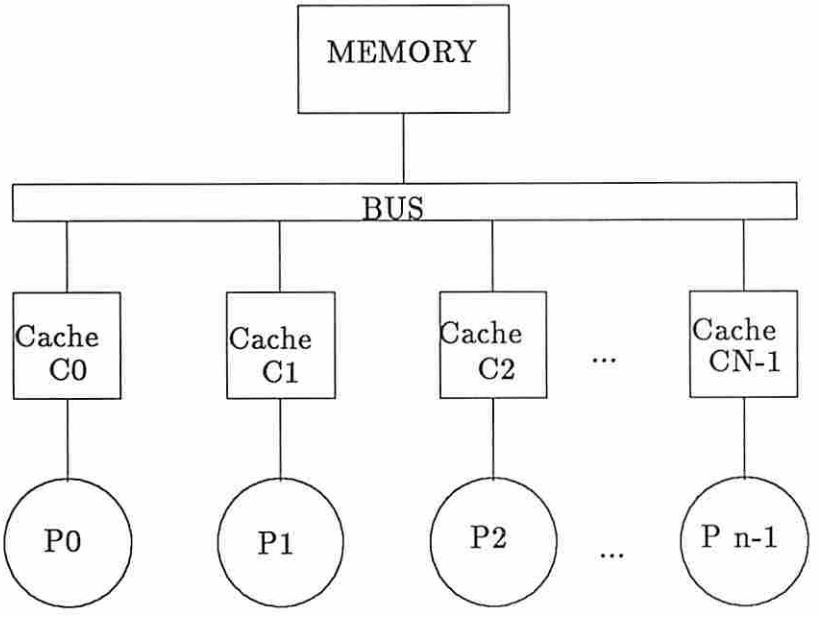


Figure 3.1: *A bus–based, cache–based system as proposed by Rudolph and Segall, 1984.*

A single bus connects all private caches of the proposed system (see Figure 3.1). The block size used by the caches is one word. Using a block size of a single word simplifies write transactions, since any single write always updates the *entire block* to a value which reflects the most recent write. The system employs a write-back scheme, which provides for a write-back whenever the block is accessed by a processor which misses at the cache and therefore causes the block to be placed on the bus. A block in a cache may be in one of three states: I, R, L. State I is the invalid state; in this case, the read path points to the main memory. State R signifies that the cache is a keeper of the block, and state L means that the cache is an owner of the block.

A block which is in the invalid state has an implied read path which points to main memory. A read miss causes an access to main memory. However, if there is a cache with an L copy of the block (owner), then the main memory request of the processor that missed on the read is interrupted by the owner. On the next bus cycle, the owner updates main memory. The interrupted processor

re-issues the read to the block. While main memory is not actually a keeper at all times, it appears that way for any cache processing a read miss.

In order for the protocol to satisfy conditions A, B, and C of Section 3.1 (and therefore be sequentially consistent), three issues must be addressed:

1. Does each processor perform read and write accesses in program order?

2. Does a writing processor stop issuing another access before the word updated by the write has become accessible to *all* processors?

3. Does a reading processor stop issuing another access before the data value that it receives by the read has become accessible to all other processors?

Question one is not addressed in the paper [51], but it is safe to assume that it is upheld.

When a cache block is in one of the three possible states of invalid, keeper, or owner, five transactions may affect it, namely: read, write, replace, bus read, and bus write. Of these we are concerned with all transactions that either follow or modify any read path. A read path is followed in only one case: when the block is not present in the cache and a read is executed. In this case, the read path implicitly points to main memory and is performed at main memory (even though a read may be interrupted once to update main memory). Once the read is issued (when it is first placed on the bus), the value returned is equally accessible by all other processors; hence, the read is performed globally.

When a block is not present in a cache i then the block's associated read path is modified in two cases: (1) A write is issued by processor i, the copy is fetched from a keeper (either an owner cache or main memory), and all read paths of all caches except cache i are modified in one bus operation to point to main memory, by means of invalidation. (2) A bus read is issued; this causes the block to be allocated to all caches. The cache block is now in state R (keeper) and the read path does not point to main memory any more. The recent modification of the block remains equally accessible to all processors.

When a block is in state R (keeper), read paths are modified under two conditions: (1) A write is issued by the processor, the block is updated, and all other caches are invalidated such that their read paths point implicitly to main memory. This is done at the same time for all processors by means of a single bus operation. The "changed" data become accessible at the same time to all processors. (2) A bus write is detected; this causes the keeper to invalidate its copy and therefore to set the read path to main memory. This occurs simultaneously to all keepers so that after the bus write, all but the owner's read path point to main memory.

When a block is in state L (owner), read paths are modified under three conditions: (1) When the block is displaced to make space for another block (i.e., it is written back to main memory): while the write-back occurs, the bus is blocked so that no access to the same block by another processor can be interleaved; once the block is written back, it is not present in the cache any more and the read path points to main memory. All other processors' caches have had read paths set to main memory all along—no modification is necessary. (2) In case of a bus read, the owner must supply the desired word and change its own status from owner to keeper. The owner places the block on the bus, and all other caches copy it and become keepers. Again, the block becomes accessible at the same time for all processors. (3) In case of a bus write, the owner must supply the valid block and change its status from owner to invalid. The read path to the block is changed to point to main memory. All processors' caches, with the exception of the one which caused the bus write, keep their read paths pointed at main memory. But, now a different owner would interrupt a bus read or write and supply the block to main memory. This change in ownership is transparent to the other caches, and they all remain equally capable of accessing the block in question.

The cache protocol has not been described in detail, but if the rules established in Section 3.1 are used, it should still be evident that the scheme

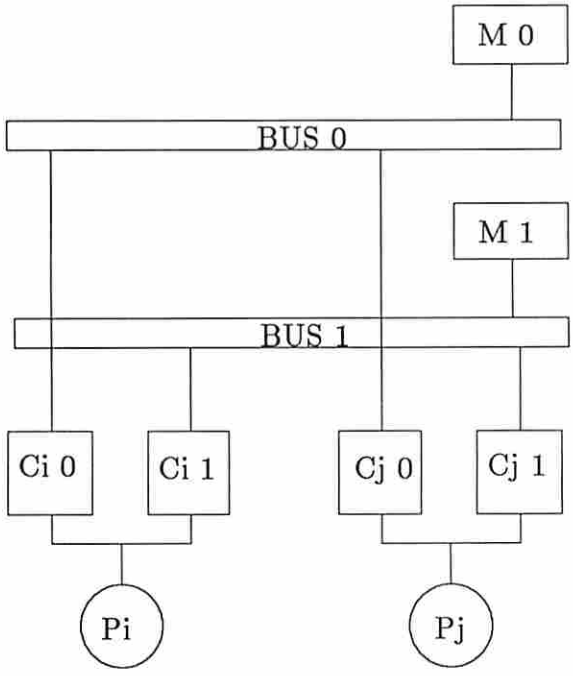


Figure 3.2: *A bus–based, cache–based system with interleaved caches and buses.*

described by Rudolph and Segall is indeed sequentially consistent. Every time a block changes state, this change is "visible" by all processors at the same time.

In the same paper, the authors propose a system with multiple buses and caches connected to an interleaved memory system (see Figure 3.2). Within each bus, connected cache, and memory module, the behavior of the system is equivalent to the one in Figure 3.1. In particular, changes of state are atomic, since every processor "watches" all buses. Therefore, according to condition 3.1, the condition for sequential consistency is simply that all accesses must be *performed* in program order. This restriction may preclude certain *apparent* advantages of the described architecture/protocol. For example, a processor P_i may not issue two subsequent writes, $M(x) \leftarrow D1$, $M(x+1) \leftarrow D2$, concurrently, even though memory locations $M(x)$ and $M(x+1)$ always imply two different caches, buses, and memory modules (because the cache block size is equal to one word). A possible incorrect outcome could be that a processor P_j reads $M(x+1)$ and $D2$ is returned before $M(x)$ has been updated to $D1$. If the

system is sequentially consistent, then the program may have been written with the reasoning in mind that if $D2$ has been generated (by P_i), then certainly $D1$ must have been generated (also by P_i). But, this will not necessarily be the case, since the two writes were not ordered, and the write to $M(x)$ may have been delayed because of bus contention.

For the same reasoning as given above, there may not be independent FIFO access queues associated with each cache, since the sequencing of accesses between queues cannot be controlled. Hence, the interleaved cache system described by Rudolph and Segall is sequentially consistent by token of the argument made for the non-interleaved system, with the restriction that a processor's previous access must have been globally performed before the next access may be issued. (In [51] it is never proposed that two or more accesses by the same processor can be serviced concurrently by different caches.) We note that an analysis of the system based on condition 3.1 allows us to identify clearly what can and what cannot be done at each stage of the protocol.

3.3.2 A table-based protocol

In [18] a multiprocessor architecture is described, that does not rely on a single system bus to propagate all accesses (see Figure 3.3). A global table, in the vicinity of main memory, keeps track of the states of all cached blocks. A block may either be in a shared state (RO), or in an exclusive state (RW). These states directly correspond to our keeper and owner states. If a block is in the RW state, then the table indicates who the owner is. A block without a corresponding entry in the table is owned by main memory. A bit, associated with every block frame in cache, indicates the state of the block as well.

All data transfers are propagated over an interconnection which allows several transfers to be in progress at one time. The interconnection can provide much greater bandwidth than a single-access bus as proposed in conventional architectures. A single-access, high-speed bus propagates *all* invalidations to *all*

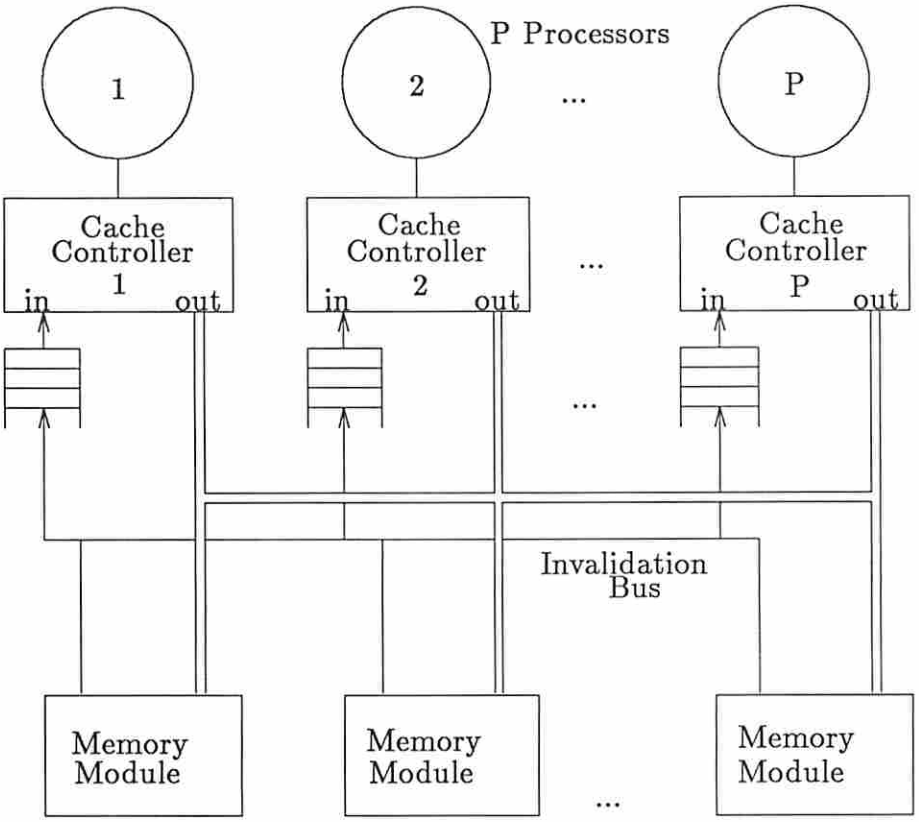


Figure 3.3: *A tightly–coupled, cache–based system with a single invalidation bus, Dubois and Briggs, 1982.*

caches simultaneously. Invalidations are buffered in a FIFO buffer at each cache where they wait to be serviced.

All read misses first interrogate the table for the state of the block. If the block is either not cached or in state RO, then the requesting cache obtains a copy of the block from main memory. If the block is in state RW, then the owner is forced to write-back the copy to main memory, and the state of the block is changed to RO. Subsequently, the cache which triggered the table interrogation becomes a keeper of the block. A write proceeds immediately, if the writing processor's cache owns the block (i.e., the block is in state RW). A write by processor P_i causes a table interrogate if it misses at the cache or if its copy of the block is in state RO. The table interrogate can have three possible outcomes:

1. No cached copies of the block exist. The entry is marked RW, with the owner as P_i's cache. A copy of the block is transferred from main memory to P_i's cache.

2. The block is marked as RO at the table. All copies of the block are first invalidated. The block is marked as RW and a copy of it transferred from main memory to P_i's cache.

3. The block is marked as RW at the table, and P_j's cache is indicated as the owner. P_j's cache is forced to write back a copy of the block to main memory and is then invalidated. The table entry is modified to indicate that P_i's cache is the new owner of the block. Then a copy of the block is transferred from main memory to P_i's cache.

The scheme is very similar to the snoopy cache scheme, in so far as the invalidation bus guarantees serialization and atomicity of updates. However, the processor to memory interconnection provides greater communication bandwidth and is not burdened with the broadcasting of invalidations. If, for the time being, the fact that invalidations can be buffered is ignored, principles A, B, and C (from Section 2) can be applied to all possible read and write situations to show that the

system remains sequentially consistent. However, if we consider the possibility of buffering invalidations at each cache, the system may not remain sequentially consistent, because invalidations can modify the read paths of the caches to a block. If the invalidations are buffered and an access is considered completed when all invalidations are buffered, it is possible that a cache i may process an invalidation before a cache j does so. This violates both principles B and C. The write which caused the invalidation is allowed to proceed after all invalidations are buffered; at this time, however, the write has not been globally performed (i.e., it has not been performed with respect to processor j). By the same token processor i may not proceed to access the invalidated block, since the write which caused the latest update has not been performed with respect to processor j.

Buffering invalidations, though, is a very appealing technique—must it be prohibited under all circumstances? By applying principles B and C cleverly, it is indeed possible to allow the buffering of invalidations. If it is possible to assume that an invalidation is performed, for all practical purposes, as soon as it is latched in the buffer of the destination cache, then the system remains sequentially consistent, provided that the cache gives the invalidation buffer priority over processor requests; in this case invalidations can be considered performed as soon as they are latched in the buffer. This is true since any subsequent accesses by the processor will *always* be serviced after the invalidation.

Above we have shown that principles A, B, and C can be applied to show protocols correct. In many cache coherence protocols it is not easy to determine whether these principles have been upheld—a thorough understanding of the protocol and the hardware is often necessary. Nonetheless, applying principles A, B, and C is usually much easier to test a protocol's conformance to sequential consistency than using any type of *ad hoc* method. On the other hand using more formal methods is likely to require even better understanding of the intricacies of a cache coherence protocol.

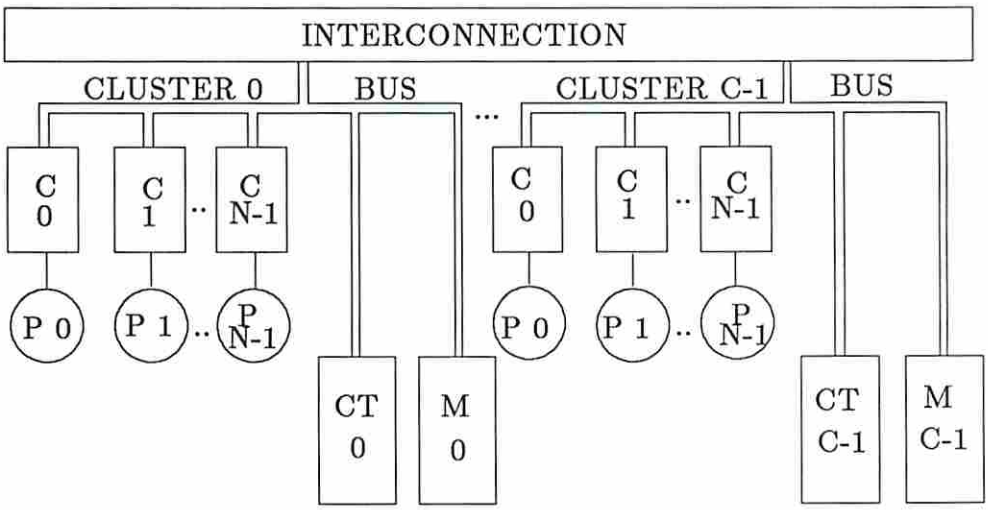


Figure 3.4: *A cluster–based multiprocessor with private caches.*

3.4 A System without Atomic Updates

In this Section an architecture and cache coherence protocol are described which do not provide for atomic updates (i.e., instantaneous setting of all read paths by a single broadcast) of data. The architecture is depicted in Figure 3.4. C clusters of processors are connected via a generic interconnection, which may consist of a crossbar, a hierarchy of buses, a loop, or any type of packet switched network. Within clusters, processors are connected by a single bus. Each cluster contains N processors. Each processor has a private cache that may contain any type of data (including shared writable data). Main memory is interleaved on high-order address bits between clusters and on low-order address bits within each cluster. Assume for simplicity and without loss of generality that each memory in a cluster is one single hardware entity. Each memory module contains M words, yielding a total physical memory space of $C \times M$ words. All processors' caches have the same capacity of K blocks, where each block consists of B words. All caches maintain a flag for each block to indicate whether the cache is an owner or a keeper of the block. A *dirty* flag, to indicate that the block has been written to, is not necessary, since all owned blocks are implicitly dirty.

BLOCK 0	K/O	READ PATH
BLOCK 1	K/O	READ PATH
BLOCK 2	K/O	READ PATH
BLOCK 3	K/O	READ PATH
BLOCK $\frac{M}{B} - 1$	K/O	READ PATH

Figure 3.5: *Format of the Coherence Table (CT).*

A coherence table (CT) is associated with each memory module and is randomly addressed; its size is M/B words. The format of the CT is shown in Figure 3.5. Each block of a memory module M_i has a corresponding entry in CT_i. A CT entry consists of a flag and a field. The flag indicates whether the block is owned or not owned, and the field uniquely identifies the keeper (an owner is also a keeper) where the block may be found. If the keeper is a cache, then the field entry consists of that cache's unique ID, c. If several caches are keepers, then the read path may point to any one of them. If the keeper is main memory, then the field's entry is blank (i.e., all 0s). The individual memory module address need not be specified, since it is implied by the block address.

3.4.1 Memory accesses

In the following, $M_i(x)$ indicates word x of memory module i, where memory module i is contained in cluster i. Five basic events (read hit, read miss, write hit, write miss, and block replacement) must be accommodated. The protocol uses write-back. Note that the following applies to accesses to shared writable data only. If processors or caches can distinguish shared writable data from local data then some rules can be relaxed. For example, processors need only be blocked from making *shared writable data accesses* while a cache miss to shared writable data is beeing processed—accesses to local data can proceed at any time.

1. READ HIT: The read is performed globally as soon as it is processed by the cache controller. This is the case because a cache can never hit on a value for which the write has not been globally performed. No external references need to be made.

2. READ MISS: A cache allocates a block upon a read miss and becomes a keeper. A read miss on $M_i(x)$ first blocks the reading processor. A request is sent to CT_i. The entry for the block which contains x is checked at CT_i, and then the access to that entry is blocked until it is later specifically released. (Any access to a blocked table entry may either be queued or rejected and retried later by the requesting processor, depending on the implementation.) The entry in CT_i specifies which memory device is presently a keeper or an owner.

   (a) If the block is owned by a cache, then the table is modified to show main memory as a keeper, the block at the owner's cache is relabelled as a keeper's copy, and a copy of the block is forwarded to the cache that experienced the original read miss. The block is also written back to main memory. Upon completion of the transaction, the CT entry is released.

   (b) If the copy pointed at by CT_i is a keeper's copy, then it may simply be copied over from the keeper (main memory) to the new keeper. When the new keeper receives the copy of the block, it sends a signal to the CT to release access to the blocked entry.

3. WRITE HIT: Two possibilities exist if a write hit occurs at cache j while accessing word $M_i(x)$: the cache is either a keeper or an owner of the block.

   (a) If the cache is a keeper, it then has to become an owner before the write can be processed at the cache. The CT_i is accessed, the block is locked and the pointer in CT_i is modified to indicate that the block is owned by cache j (i.e., the read path points to j). Then invalidations of

the block are sent to all clusters, through the cluster interconnection. Cache j changes the status of the block to owner. When all caches have been invalidated (in any sequence), the block entry in C_i which refers to the block which contains x is made accessible to other requests again. At this point P_j may also proceed to issue another memory access.

(b) If the cache is an owner, then the write can simply be processed by the cache controller; no other external activity is required. The processor may proceed as soon as the cache has modified the word.

4. WRITE MISS: A write miss by processor P_j to word $M_i(x)$ first causes processor P_j to block. Then CT_i is accessed for the block information. The read path is followed to the keeper/owner of the block, and a copy of it is forwarded to Cache j. The entry is modified to indicate that cache j is now the owner of the block. As usual, the accesses to the modified entry of CT_i are disallowed until a later time. Originally, the block was either owned or kept.

   (a) If the block was owned, previous to the write miss, then when a copy is fetched from that owner, that cache must also be invalidated (this can be done in one operation).

   (b) If the block was kept by several caches before, then not all keepers are explicitly known, and an invalidation must be broadcast to all caches. Once all previous copies of the block are invalidated, requests may be successfully made again to CT_i concerning that block. When cache j receives a copy of the block, it may proceed with the write, and processor P_j may continue issuing memory accesses when all other copies of the block have been invalidated.

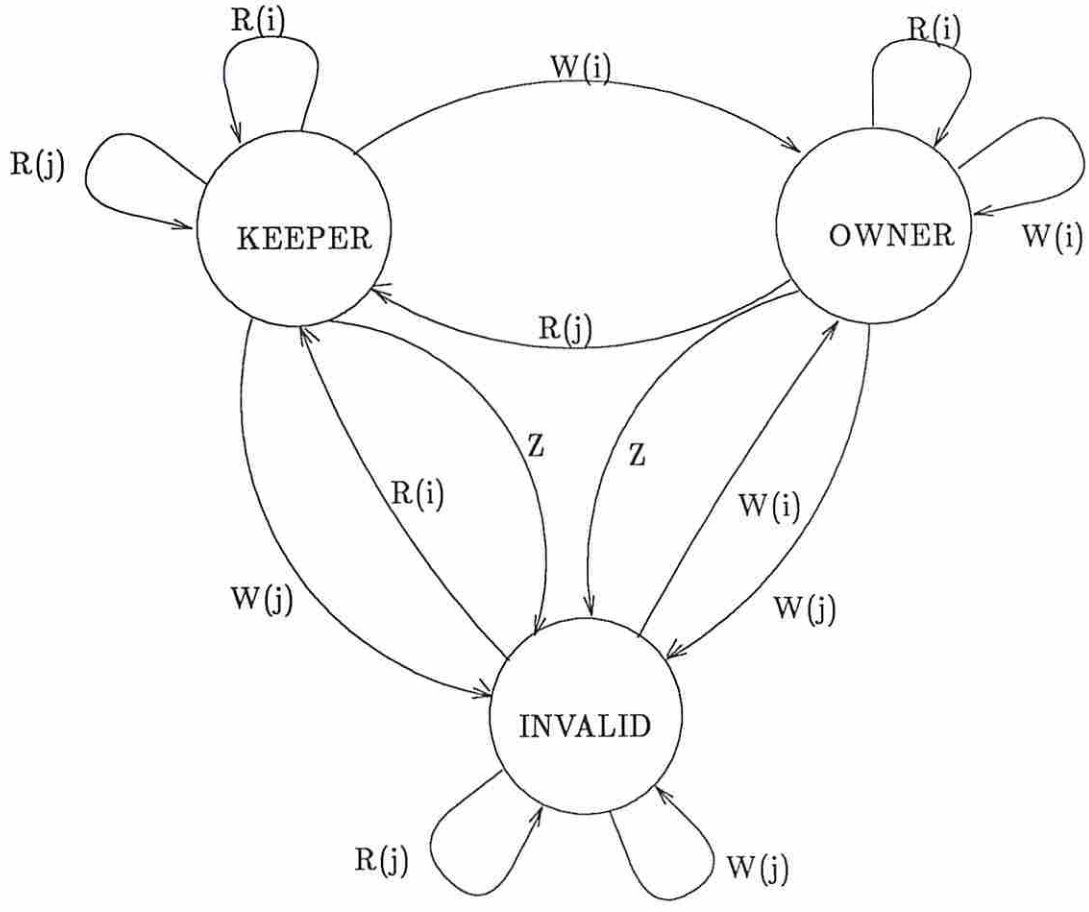
5. BLOCK REPLACEMENT: If the cache is a keeper, then the block may be removed without further action. A cache becomes a keeper by either

fetching the block from main memory or by fetching it from a previous owner. In either case main memory maintains a valid copy. This is true because when a block is copied from an owner, it is also written back to main memory. If the cache is an owner of the block, then procedure (4a) is followed with the exception that no actual word is updated. Main memory becomes the new owner. In the case that an outside access to a block is attempted which has been invalidated (this is possible since the CT may have a read path point to a cache which later replaces that block) the access must then be forwarded to main memory which is, in this case, guaranteed to contain a valid copy of the block.

A state transition diagram is shown in Figure 3.6.

At most times it is indistinguishable whether a read path points at a cache or main memory. Main memory, however, need not keep track of whether the copy of a block that it retains is stale, a keeper's copy, or an owner's copy. This is true since this information is already kept in the appropriate CT. Practical systems will incorporate the table into main memory, such that a table access and a memory access are the same operation.

The reader may verify independently that the conditions for sequential consistency, as defined in Sections 3.1, are indeed upheld. The reason is that only one update for a particular block may be in progress at any time. The fact that the actual updating of data occurs at different times for different copies is irrelevant, since all accesses may either return an "old" copy of data, or must wait until the "new" data is available to *all* processors. Invalidations must in many cases still be broadcast to all caches. However, a scheme, similar to that in [12], to filter out useless invalidations could be implemented. In a system, though, where the amount of shared, writable data is relatively small or where accesses to such data exhibit good processor locality over time, a very large number of processors may be interconnected efficiently, using a large shared memory space.



R(i)=Read by processor i
W(i)=Write by processor i
R(j)=Write by processor j, $j \neq i$
W(j)=Write by processor j, $j \neq i$
Z=Block replacement

Figure 3.6: *Transition diagram of data block state in cache i. A cache may be in one of three states w.r.t. a block: (K)eeper, (O)wner, and (I)nvalid.*

3.4.2 Indivisible memory accesses

Sequential consistency guarantees that the order in which accesses are performed with respect to different processors is never reversed. To implement synchronization primitives such as Test&Set not only must an order between accesses be preserved, but the successive accesses must also be performed *indivisibly.* That is, no other access to the operand may be interleaved between the Test and Set portions of a Test&Set access.

Indivisible accesses are easily implemented in bus-based systems, since a processor may hold the bus until its entire atomic access has been performed. While the bus is being held by the processor, no other processor can physically access the memory module or cache in question. Additionally, ownership schemes are easily implemented in bus-based systems—when such protocols are used, atomicity of accesses is easily implemented. In the system proposed in Section 3.4, preventing other processors from making global accesses is not possible, since there exists no unique bus which can be blocked by the processor wishing to perform an indivisible sequence of accesses. However, blocks may be owned and ownership cannot be released without the owner's consent. Switching ownership is also always serialized by the CT. Hence, an atomic access can be correctly executed if the block upon which the access is to be performed is owned by the accessing processor. For all practical purposes, with respect to synchronization accesses, the system reverts to an architecture with a single copy per block. However, the single copy may be located wherever it is most convenient.

The sequence of events which occurs if two processors, P_i and P_j, access a binary lock (semaphore) L, which may have states of 0 and 1, is described here. Initially $L = 0$, which indicates that the lock is open. P_i obtains ownership of the lock by performing a dummy write operation which does not alter the value of the lock. Once the lock is owned by P_i it successfully executes a Test&Set on it, so that $L = 1$. P_j wishes to obtain the lock, it performs a read on it, with the result that both cache i and cache j become keepers of the lock. P_j then spins on the lock, which is contained in its local cache, and therefore no global activity

results. After some time P_i wishes to release the lock. Since this involves a write to L, P_i first obtains ownership of the lock and then resets it to 0. On the next Test of the lock, P_j misses at the cache, so the cache then recopies the unlocked copy, making both caches keepers of the block. P_j finds that the lock is open and proceeds to issue the dummy write, which makes it the owner of the lock. Once P_j is the owner, it may proceed with a Test&Set operation. The following is a simple spin_lock algorithm.

```
Spin_lock(x)
while (temp <> 0)
  {while (temp <> 0)
   temp = (x);
   temp = Test&Set(x);
  }
```

A Test&Set operation must result in the ownership of the block. That is, it must be treated like a write operation. The only caveat is that a spinning processor should not use the Test&Set to spin on a lock after failing initially in obtaining the lock. Otherwise, several processors spinning on the same lock can cause severe system degradation by passing the lock continuously back and forth. A similar problem may also occur if several locks are "packaged" within the same block.

3.5 Isolated Caches

Cache hierarchies have recently become the focus of attention of several researchers working on cache coherence protocols. The apparent advantage of different cache levels is the ability to tailor each cache to perform best at its physical location. The closer to a processor a cache is located the faster it must be, at the price of being small. Caches physically more distant from processors must have large capacities and possibly high throughput, usually at the cost of having higher access latencies.

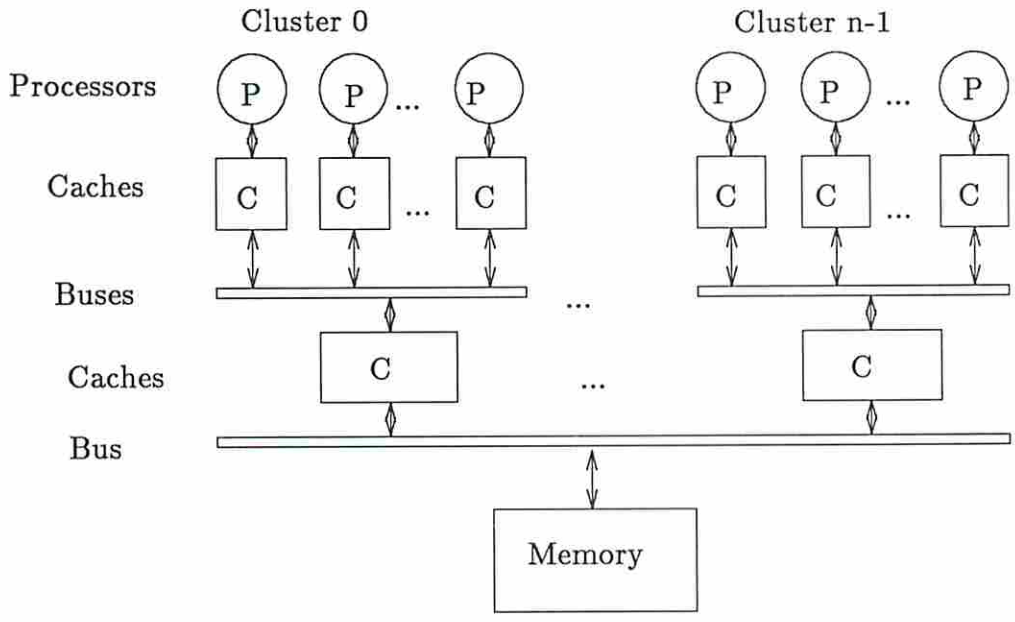


Figure 3.7: *Hierarchical cache/bus architecture.*

In [66], Wilson describes a hierarchical bus/cache architecture. In [7] and [8], Baer and Wang describe some necessary logical properties of coherent cache hierarchies. The most important property described is the *inclusion property* which manifests that a higher level cache (a smaller cache closer to the processor) should always contain data which are a subset of data contained in the next lower level cache.

The rules governing sequential consistency in cache–based multiprocessors are directly applicable to systems using cache hierarchies. Figure 3.7 shows an architecture using hierarchical caches and buses similar to the one proposed by Wilson in [66]. A processor's memory access always traverses the hierarchy until it can be satisfied. Caches can either be data–less nodes in the read path of an access or can be keepers or owners of words (blocks). To implement efficiently correct ordering and consequently coherence, the *inclusion property* should be upheld. Properties A, B, and C must be maintained in hierarchical cache multiprocessors to preserve sequential consistency. How this is accomplished specifically depends on the underlying architecture.

3.5.1 Isolated caches-processor modules

It may often be desirable to allow the processor and cache controller to temporarily isolate themselves from the outside environment. This is particularly attractive if processor and cache are contained on a single chip which is capable of significantly higher execution speeds when it is not necessary to immediately respond to every incoming coherence signal. Since sequential consistency is defined on the basis of *order* rather than timing it is possible for subsystems to temporarily isolate themselves from the remainder of the multiprocessor. This capability is not limited to hierarchical cache systems but has more application in such architectures. In the following we assume a system and a cache coherence protocol similar to the cluster–based architecture of Section 3.4. The procedure to isolate a single processor and its closest cache memory is discussed—whether there exist more cache levels "under" the isolated cache is irrelevant assuming that the "underlying" system maintains the prescribed protocol.

Assume that a FIFO queue buffers external (coherence) requests. Only signals and data which are not directly or indirectly caused by the processor itself (such as acknowledgments or requested data blocks) are considered because processor-caused signals are "expected" by the processor and the processor or cache controller will usually wait for them while idling. The processor and the cache normally interact without "outside" communication. Coherence requests are buffered in the queue and the queue is serviced either upon the cache controller's initiation or when the queue controller signals the cache controller that it needs to be serviced (see Figure 3.8).

Another processor's memory access request is always the original cause for an incoming coherence request (such as an invalidation). Such a memory access can sometimes be considered performed with respect to the local processor once its resulting coherence request is buffered in the queue. In other instances the request must be processed by the cache controller before the memory access can be considered performed with respect to the local processor. The term *performing a coherence request with respect to a processor* will be used to indicate

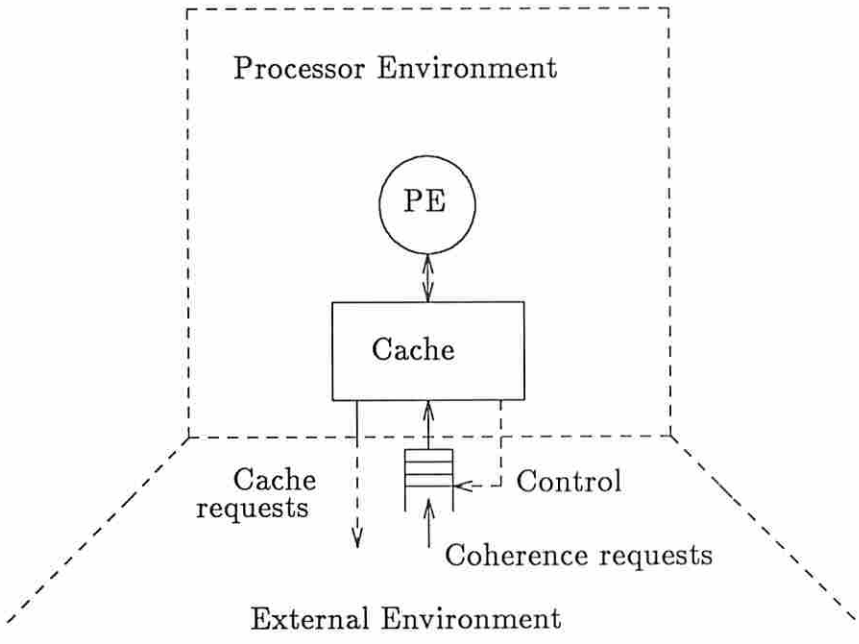


Figure 3.8: *Isolated processor and cache environment.*

that the associated memory access has been performed with respect to a processor. Four types of coherence requests are possible in the discussed protocol:

1. Invalidate block.
2. Send copy of block.
3. Send copy of block and then invalidate block.
4. Modify block. (This request is not part of our protocol but could be with only small modifications.)

The system remains sequentially consistent under the following conditions.

1. A cache access either resulting in a miss or any write to a non-owned copy of a block requires suspension of the access until all pending requests in the queue are serviced in FIFO order. After servicing *all* pending requests the original access must be retried. Complete retrial of the access is required because the servicing of the pending requests may have changed the

local status of the accessed block. (For some access types and protocol combinations, a complete retrial may not be necessary.)

2. Coherence requests which are *modifications of storage* or *modifications of a read path* can be considered performed with respect to the local processor as soon as they are buffered in the queue. Such requests are, for example, broadcasts of blocks or invalidations of blocks.

3. Coherence requests which are (1) *following a read path and reading data* or (2) *requesting/releasing ownership* cannot be considered performed with respect to the local processor until they are serviced by the cache controller. Such requests are, for example, a request for a copy of a block or a request to release ownership of a block.

4. Whenever the queue is serviced, *all* queued requests must be serviced in the arrival order without interleaving newly incoming requests or processor memory accesses.

That sequential consistency is maintained if all coherence requests are of the second type should be clear because the above condition (3) conforms to the previous conditions for sequential consistency. It is necessary, though, to demonstrate that taking advantage of the above condition (2) does not violate sequential consistency. It is clear that condition (2) violates the definition of *performing an access* with respect to a processor. For example, if an invalidation is buffered rather than serviced then the processor can subsequently access an "old" copy of a block while in fact the write which caused the invalidation has already been considered globally performed. However, this occurrence is not detectable because it guarantees the same outcome as in the case when the processor read occurs before the invalidation. Hence, the behavior of the system is indistinguishable from that of a system strictly adhering to rules A, B, and C.

Let us assume that a processor–cache system experiences the following sequence of events:

1. The processor and cache experience an number of hits while the queue is filling up with coherence requests.

2. For some reason (it can be any reason) the queue is to be serviced.

3. The processor blocks (at least with respect to shared-writable memory accesses).

4. All the requests in the queue are serviced in their original arrival order.

5. After the queue is empty, if there is a pending miss by the processor (which may have originally triggered the servicing of the queue) then it is issued and globally performed.

6. If any more requests have queued up while going through the previous step then those requests are serviced in their arrival order.

7. The processor is unblocked and performs a series of accesses $a_1, \ldots, a_n$ which can all be locally satisfied by the cache.

8. At the same time a number of coherence requests $r_1, \ldots, r_m$ are queued at the buffer.

9. The buffer is serviced again *before* the processor (re)issues another access a_{n+1}.

Without a buffer the servicing of accesses $a_1, \ldots, a_n$ and requests $r_1, \ldots, r_m$ would have been interleaved in some manner rather than sequenced in the order $a_1, \ldots, a_n, r_1, \ldots, r_m$. However, since processor speed is never taken into consideration it could have been possible for a faster cache and processor to issue and perform accesses $a_1, \ldots, a_n$ in the time it actually took to only issue and perform access a_1. In which case the interleaving resulting from buffering would have occurred. Hence, that particular interleaving *does* conform to sequential consistency. Note, that this is true only because the processor and cache are "isolated" from the rest of the system while not servicing the buffer and because

the pending requests are serviced one-after-the-other in the arrival order without interleaving any processor accesses. If "isolation is broken" for any reason (inter-processor interrupts, for example) then the queue must always be serviced before the processor can react (i.e, service an interrupt).

Two approaches exist to manage coherence requests which are *not* performed with respect to the local processor once buffered. (1) Either they are buffered and treated like all other requests, or (2) they trigger the immediate servicing of the queue. In the first case remote processors may be kept waiting longer than is practical; in the second case the local processor and cache controller may be interrupted more frequently than is desirable. The architecture, coherence protocol, and amount of data sharing will impact the efficiency of each approach. Since a full queue must trigger the servicing of the queue, a bufferless system can be categorized into a system with a single-element queue (i.e., a request latch).

3.6 Conclusion

In this Chapter a simple condition for sequential consistency of multiprocessor systems was applied to cache coherence protocols. Sequential consistency is a desirable property of general-purpose multiprocessors in which data can be shared; multiprocessors designed to run multitasked programs written in concurrent languages should also adhere to sequential consistency.

To illustrate our approach, we have demonstrated the correctness of several protocols that have been published. These protocols rely on a full broadcast system (either a system bus or an invalidation bus) to propagate updates atomically. Finally, we have described and verified a protocol in which updates are not atomic but propagate from cluster to cluster in a cache-based machine. We have shown that the resulting multiprocessor is not only sequentially consistent but can also execute spin-locks based on indivisible sequences of loads and stores efficiently.

The fact that sequential consistency can be implemented in hierarchical-cache systems was demonstrated in the last Section of this Chapter. Furthermore, it was shown that a system can remain sequentially consistent while processor-cache pairs can temporarily isolate themselves from the system to be able to work uninterruptedly. Such systems are not strictly coherent but they are generally coherent and remain sequentially consistent.

The contribution of this Chapter is twofold. First, we have shown how to apply a simple condition to the verification of complex coherence protocols; the condition permits us to analyze every step of the protocol. Second, we have demonstrated that *logical* problems associated with large-scale cache-based systems can be solved to provide sequential consistency and reliable execution of indivisible sequences of loads and stores; these large scale systems are relieved from the condition that *all* processing elements be tied to an invalidation or system bus.

Chapter 4

LOCKUP–FREE CACHES IN MULTIPROCESSORS

The performance of shared memory multiprocessors can suffer greatly from moderate cache miss rates because of the usually high ratio between memory-access and cache-access times. In this Chapter we propose a cache design in which the handling of one or several cache misses occurs concurrently with processor activity. Concurrent miss resolution in multiprocessor caches must function in conjunction with the system's synchronization hardware and cache coherence protocol. Through performance models, we identify system configurations for which concurrent miss resolution is effective. Compiler techniques to take advantage of the proposed design are illustrated at the end of the Chapter.

4.1 Introduction

Cache memories are commonly used to reduce memory access time for both data and instruction accesses. Caches can do this very effectively and economically [59]. In shared memory multiprocessors caches are more important than in uniprocessors because the individual processors of a multiprocessor must be connected to the shared memory through an interconnection. Increased

memory access latency and conflicts reduce the efficiency of each processor [30]. Prefetching, which can reduce the apparent access latency visible to the processors, is also more difficult in multiprocessors because of the coherence problem [12].

It is possible to design caches which do not block the processor on an access miss—they are called *lockup-free caches.* In such designs the processor may continue sending requests to the cache both for data and instructions while the cache and the main memory system are resolving previous misses. Such a scheme was described by Kroft in [37] for a uniprocessor. When processors are part of a shared memory multiprocessor, the design of lockup-free caches becomes difficult because of the added problem of maintaining cache coherence.

In this Chapter we focus on multiprocessor architectures which allow caches to be lockup-free and in which synchronization is enforced by means of "hardware-guarded" primitives. In such a system memory coherence need only be restored upon execution of a synchronization access. Because the specifics of a coherence protocol are not important as long as the above condition is met, different cache coherence algorithms can be used in conjunction with the lockup–free protocol.

4.2 Multiprocessor Caches

4.2.1 Impact of cache performance

Even small cache miss rates can have detrimental effects on the throughput of high-speed processors. As the following evaluation demonstrates, the efficiency of a processor can deteriorate rapidly when the memory access time is long or when the hit ratio is low.

Let us assume a processor system with the following characteristics:

X is the maximum throughput of the processor in MIPS (Million of Instructions Per Second) if all accesses can be resolved by the cache (i.e., a cache hit rate

of 1.0). X can often be easily estimated for a given processor architecture and instruction mix.

d is the average number of accesses per instruction execution, including instruction fetches, operand fetches and resultant stores. It is called the demand rate.

T_m is the average time to resolve a miss via main memory.

h is the average cache hit rate.

$t_{i,0}$ is the average time it takes to execute one instruction if all accesses are cache hits (i.e., $t_{i,0} = 1/X$).

If the processor blocks on every miss, the average time to execute one instruction, t_i is given by $t_i = t_{i,0} + (1 - h)dT_m$. Hence, the average performance of the system, in MIPS, is reduced to $X' = X\frac{t_{i,0}}{t_{i,0}+(1-h)dT_m}$. Dividing by $t_{i,0}$ and letting $T_m^0 = T_m/t_{i,0}$, yields $X' = X\frac{1}{1+(1-h)dT_m^0} = XF_s$. F_s is called the *slowdown factor*. It varies between 0 and 1 and the closer it is to 1, the better the processor efficiencies. In Figure 4.1, the performance degradation of a shared memory system is shown for different values of h and T_m^0. As can be seen in Figure 4.1, even a system with a relatively high hit rate of 0.98 can suffer substantially, if the average main memory access time is long relative to the cache access time. Particularly in the case of fast processors, the ratio T_m^0 is likely to be in the upper ranges shown in Figure 4.1.

4.2.2 Multiprocessors versus uniprocessors

In a multiprocessor, even if processors have large private caches, a great deal of memory to cache communication is still necessary due to the need to propagate block updates (usually in the form of an (1) invalidation, (2) fetch block sequence). Frequent updates and consequent invalidations have two effects—they strain the bandwidth capabilities of the interconnection and they lower the

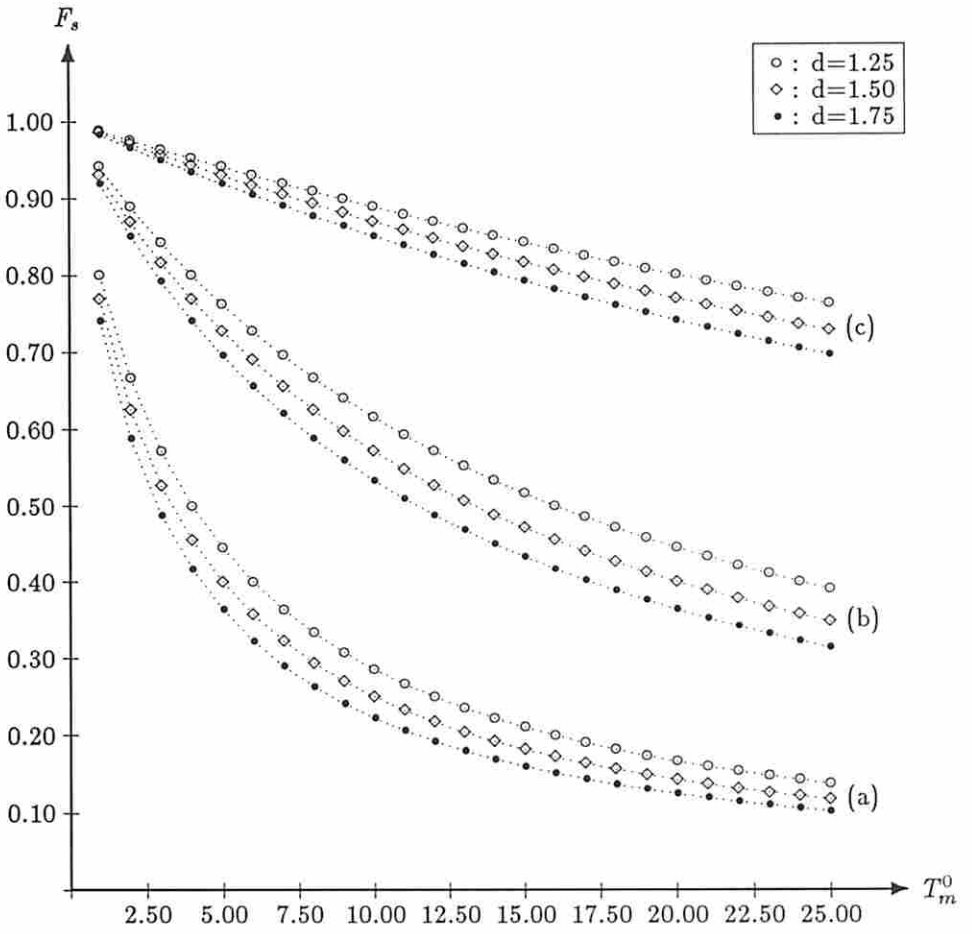


Figure 4.1: *Slowdown factor vs.* T_m^0 *for hit rates of (a) h=0.8, (b) h=0.95, (c) h=0.98.*

invalidated caches' hit rates [20]; for small numbers of processors the traffic can cause contention on the interconnection bus. If more processors are used then a bus would saturate and a multi-point interconnection must replace the bus. In either case the average main memory access latency is increased.

The penalty of large average memory access latencies can be partially avoided if processors are not constrained to wait for individual misses to be resolved before initiating another access. The use of lockup-free caches in multiprocessors is restricted, though, by the fact that coherence problems can arise when accesses are performed out of program order. This may be the case in a lockup-free cache if, for example, a miss is followed by a hit, with the result that the access which hits is performed before the access which misses because of the longer time to resolve the miss. The restriction on the order in which accesses must be performed is due to the possibility of *inter-process* dependencies [19].

If a system makes no attempt to enforce *all* inter-process dependencies, and the programmer and compiler are aware of this fact, then the out-of-program order execution of accesses by a single process is allowable. (Of course intra-process dependencies must still be preserved.) In such a system it is possible to design caches to be lockup-free. In this case since simple loads and stores cannot be used to implement synchronization [19], special hardware recognized primitives, such as the `test&set` instruction, must be used to implement synchronization.

4.2.3 Restrictions on ordering

With respect to the *ordering of events* within a multiprocessor, the user (or compiler) may expect the system to either be sequentially consistent or weakly ordered. It is very difficult to implement sequential consistency in systems which allow lockup-free cache operation on *all* types of memory accesses. To preserve causal correctness and general coherence in systems in which all accesses by the processor are resolved in program order is likely to involve unacceptable hardware overhead, or timing-sensitive software.

A weakly ordered system, on the other hand, is not sequentially consistent and assumes three types of shared data.

1. Instructions, private, and non-writable data can be accessed and cached by all processors in any possible order. Since non-writable data are never modified, no inter-process dependencies can exist on such data. This is also true for private data which are only modified and read by one processor.

2. All other ordinary shared writable data can only be modified in *mutual exclusion.* These are data used to transfer information from one process to another.

3. Synchronization variables are data used to enforce mutual exclusion on write accesses to ordinary shared writable data. Synchronization variables are hardware recognizable as such.

Data of type (1) pose no difficulty and will not be further discussed. Data of type (2) are user/compiler generated. Accesses to such data must be protected by critical sections or semi-critical sections [21]. In the first case data may only be read or written by one processor at a time—the processor which has gained access to the appropriate critical section. In the second case, several processes are allowed to read the same data at the same time.

Accesses to data of type (3) are synchronization primitives such as `test&set` operations. Since data modified within a critical section cannot be read by another processor, while the modifying processor is still executing the critical section, the order in which the data are modified within the critical section is immaterial. The only constraint for correctness is that all updates have properly propagated when the critical section is exited. However, a processor is not "aware" whether it is presently executing a critical section or not. Since critical sections may be nested or overlapped, keeping track of critical sections by the processor is not simple. For correctness, though, it is sufficient that all accesses of type (2) have propagated and completed, before an access of type

(3) can complete. If a synchronization variable access is encountered, either a critical section is entered into or one is exited from—in either case all previous accesses must have been performed.

We summarize this section by defining four properties that must be maintained for a weakly ordered system to remain correct:

P1: Intra-process dependencies must be observed at all times.

P2: All modifications of type (2) data must be performed from within critical sections. The compiler or the programmer must ensure that no two or more processes can be in the same critical section at the same time.

P3: Reliable synchronization mechanisms must be implemented. Traditional memory coherence with respect to type (3) data must be enforced to ensure that the multiprocessor is sequentially consistent with respect to synchronization accesses.

P4: Memory coherence of type (2) data must be restored at synchronization points (when an access to type (3) data is made).

4.3 Lockup–free Caches in Weakly Ordered Systems

Most of the time the cache responds to processor requests of type (1) and type (2) data. With respect to such data, the operation of the cache is equivalent to the operation of a uniprocessor lockup-free cache. Kroft described the implementation of such a cache in [37]. A basic overview of Kroft's principle of operation is given here; for more details the original paper should be consulted.

Multiple misses are resolved by storing information about each and then forwarding the miss request, packaged along with some vital return information, to the main memory. This is accomplished with the following in mind.

- If a missed and to-be-returned block is to be allocated, space must be reserved in the cache for that block[1] and, if necessary, a replacement must be made.

- Miss requests must be tagged such that:

  1. The word of the block which caused the miss is known.
  2. The functional unit which the word is to be forwarded to is known.
  3. The cache frame which is reserved for the block is known.

Most of the above qualities are implemented by a set of associatively accessible registers, called MSHR registers (Miss Information/Status Holding Register), which keep track of the status of all pending misses.

4.3.1 Enforcing the rules

4.3.1.1 Enforcing P1

Property **P1** is well understood and is often implemented in pipelined processors. The three basic intra–process hazards RAW (read-after-write), WAR (write-after-read), and WAW (write-after-write) need to be detected and and avoided. Upon detection of a hazard the processor blocks and only issues the access causing the block once the hazard has been resolved.

4.3.1.2 Enforcing P2

Property **P2** is enforced by the compiler.

[1] In multiprocessors, the early reservation of a cache frame for a to-be-returned block may not be prudent because the case of an invalidation of a block arriving while the block is not present but allocated must then be accommodated. We assume that a cache frame is only allocated once its target block is actually present.

4.3.1.3 Enforcing P3

The reliable implementation of synchronization involves both hardware and software issues. The hardware must ensure that all accesses to type (3) data are *always* memory coherent and sequentially consistent. This means that system–wide a read of type (3) data always reflects the most recent update of such data [12], and that, within a process, accesses to type (3) data are always issued in program order. The instruction set should include indivisible read–modify–write instructions which simplify synchronization algorithms and make them easier to debug and verify. Programmers may choose to let the compiler generate synchronization accesses by using high–level–language commands to specify critical section entries and exits. If, however, programmers are allowed to implement their own synchronization algorithms, using type (3) data, then programming errors can lead to the failure of type (2) data coherence. A compiler can attempt to detect such errors but can never guarantee property **P2** if it does not have exclusive control over synchronization.

4.3.1.4 Enforcing P4

An established coherence protocol can be used as the basis to enforce property **P4**. The same protocol is used for all three types of data, except that misses on type (3) data block the cache. The "restoration" of coherence of type (2) data upon synchronization points is accomplished by waiting for all pending activities on such data to be resolved. In the case of read misses this means that pending blocks must reach the cache. In the case of write operations, all modifications of words issued before the synchronization point must be completed. Depending on the specific coherence protocol used, this condition may be met automatically or may require some additional control.

4.3.2 Property P2 and established coherence protocols

While a weakly ordered system ensures that multiple processors cannot attempt to modify the same word concurrently, it is still possible that multiple processors write different words belonging to the same block concurrently. Such multiple writes must be serialized as to leave only one copy or multiple copies of the block which reflect(s) *all* the updates. This problem must be addressed in *all* cache coherence schemes but the degree of modification needed to make caches lockup–free depends on how a particular protocol solves this problem.

Two methods to serialize write operations are commonly used either separately or in combination. The first method serializes write permission by allowing only one cache at a time to contain an exclusive, modifiable copy of the block. The second method serializes actual write broadcasts by using a hardware serializing device, such as a bus or central controller.

4.3.2.1 Write permission serialization

Write permission serialization is commonly used in invalidation–based protocols. Using a distributed scheme, all caches are connected to one or multiple bus(es). Each cache observes all bus activities (snoops) and takes appropriate actions. In its simplest implementation a block in cache is in either state invalid, read–only, or read–write. A write access may only proceed if the block is present in state read–write. A write miss causes the block to be loaded either from cache or from memory. All other copies of the block in other caches are invalidated. Both the *Synapse* and the *Berkeley* protocols are approximate examples of this type of protocol [25,34].

Variations of this type of protocol can also be adapted to cache coherence schemes based on centralized directories [5]. In either case, *write permission serialization* is directly compatible with lockup–free caches. The problem which must be addressed when using such protocols with lockup–free caches is the proper handling of concurrent modifications and pending misses of the same

block at different caches. More precisely, the protocol should be correct for the following occurrences:

1. A read miss on a block for which a read miss is already pending at a different cache.

2. A read miss on a block for which a write miss is already pending at a different cache.

3. A write miss on a block for which a read miss is already pending at a different cache.

4. A write miss on a block for which a write miss is already pending at a different cache.

In case (1) the accesses can be for the same word in the block; in cases (2, 3, and 4) property **P2** excludes that possibility. Hence, there exist no data dependencies among the pairs of accesses and the order of execution is irrelevant. The protocol must only ensure that:

- All write operations always either update all copies of a block or update some copies and invalidate all other copies.

- Always at least one valid copy of a block remains either in cache or in main memory and the location of the copy is explicitly or implicitly known.

4.3.2.2 Write broadcast serialization

Protocols which employ *write broadcast serialization* only, are usually write–through protocols. Upon every write operation the modified word is broadcast to all caches. The broadcast is serialized via either a single bus or a central controller. Again the problem of concurrently pending misses and accesses referencing the same block exists. The conditions for correctness are the same as in the case of serializing write permissions.

4.3.2.3 Hybrid write permission/broadcast serialization

More advanced cache coherence protocols use hybrid schemes which sometimes choose to broadcast writes but can also allocate exclusive write permission to caches. The *Firefly* and *Dragon* protocols are examples of such protocols [5]. The problems encountered when adapting these schemes to lockup–free operation are the sum of the problems of the two exclusive approaches.

4.3.2.4 Serialization and multiple misses

Two problems must be addressed. Firstly, how does a cache react when a write or invalidation occurs while the cache is waiting for a miss to be resolved, and secondly how must consecutive misses referencing the same block be controlled in such a way as not to confuse the lockup–free mechanism.

The easiest way to ensure no overlapping of write attempts to a block with pending misses of the block at another cache is to provide a busy line [25]. In a bus–based system the snooping controller keeps track of pending misses and raises the busy line if it detects another cache's bus access for a pending block. The writing cache aborts its write when it detects the busy signal and retries the access later. If a central controller is used then conflicting accesses to the same block can either be buffered until previous accesses are fully completed or denied by sending a *try–again–later* message to one of the caches.

The case of consecutive misses on the same block by the same processor can be controlled in two ways. The MSHR registers can be set up to keep track of more than one miss and upon miss resolution pass data or acknowledgements to the processor corresponding to all the pending requests for the block. An easier scheme is to detect multiple misses referencing the same block and to take action only on the first miss. Subsequent misses referencing the same block are simply recycled until they can be resolved. This approach is shown in the sample architecture of Section 4.4.

4.3.3 Special cases

Two other approaches exist to alleviate the problem of concurrent write operations. (1) The cache block size can be set equal to one word and partial-word write operations are prohibited [54]. In this case the cache must simply ensure that all other copies of a block are invalidated upon a write operation. Concurrent writes to the same block are not possible due to the rules of a weakly ordered system. Problems with this approach are the lowered overall hit ratio due to the small block size and the restrictions on partial–word write operations. (2) The compiler can allocate data in such a way that data packed in one block are always modified by only one processor at a time [55]. That is, the data packaged in a block are always accessed in the same critical section. Depending on the application, such data allocation may either be difficult to accomplish or may be inefficient.

4.4 A Sample Architecture

In the following, a lockup-free cache architecture is described to demonstrate the ease with which a given architecture can be modified to be lockup–free assuming a weakly ordered system.

4.4.1 General features

The architecture relies on multiple broadcast buses and the cache coherence scheme is a snoopy–bus protocol. The selection of a particular bus is based on the block address and transactions with respect to a given block are always restricted to one bus only. In the following descriptions only a single bus and a single snooper per cache will be discussed. The operation of the "parallel" snoopers and buses is exactly the same as for the one shown. Caches operate in write–back mode. All addresses are physical addresses. (The implementation of virtual memory is not difficult as long as physical caches are supported by

translation lookaside buffers (TLBs); descriptions on how to service TLB misses would, however, obfuscate the relevant issues at hand.)

While the description assumes a trivial three–state coherence protocol, more complex protocols can also be integrated into the architecture. The basis of the given protocol is that *every* write–back protocol implements the three-state protocol but improves efficiency by assigning different "flavors" to some of the states. We ignore the possibility of write–broadcasts but note, that in a snoopy–bus, invalidation–based protocol the mechanics of broadcasting invalidations and writes are at least logically *very* similar to those of write broadcasts. The following assumptions hold:

1. Processors issue accesses to cache only if the access does not cause a hazard with respect to a not yet resolved miss.

2. Caches maintain records on pending misses, such as block address, operation type that missed, word that missed, destination of word (in the case of a read), and some other information in a set of registers called MSHR (miss information status/holding register) registers—using Kroft's terminology [37].

3. A cache blocks the processor from issuing more accesses when it has exhausted its storage capability to keep track of pending misses.

4. Blocks are either absent in cache, or present and tagged as **RO** (Read Only) or **RW** (Read Write). A **RO** block copy may be read but not modified. A **RW** block copy can be read and modified. Only one **RW** copy of a block is allowed to exist at any given time.

5. A block is fetched if there is a write miss (write allocate).

6. A synchronization access cannot proceed until all previous accesses by the processor have been completed.

7. All coherence traffic with respect to any one block is globally serialized by the bus which the block address maps to.

8. Upon replacement block frames in state **RO** are overwritten and blocks in state **RW** are written back to main memory.

The following nomenclature applies:

A destination within a processor is the logical or physical intended destination of a resolved read access. We assume that a destination is always a register in the processor. In the case of a write a destination is a processor flag which is reset once the write has been performed.

A block read request by a cache is answered with a copy of the block. If the block exists as a single **RW** copy in a cache, then that cache supplies the block copy and changes the state of its copy from **RW** to **RO**. If no cache copy or several **RO** copies of the block exist, then main memory supplies a copy of the block.

A RW block read request by a cache is answered with a copy of the block, and causes the invalidation of all other copies of the block.

4.4.2 Cache operation

A processor node consists of three interacting devices: (1) the processor, (2) the cache buffer controller, and (3) the snooper. Three main queues buffer requests between the devices. The P-queue accepts requests from the processor and the snooper and is serviced by the cache buffer controller. The S-queue accepts requests from the cache buffer controller and is serviced by the snooper. The B-queue accepts entries from the snooper and accesses the main system bus. (See Figure 4.2.) The cache buffer controller has access to the main cache buffer and the main tag directory. The snooper monitors the bus, and maintains the MSHR registers. Each MSHR is split into fields indicating the type of operation

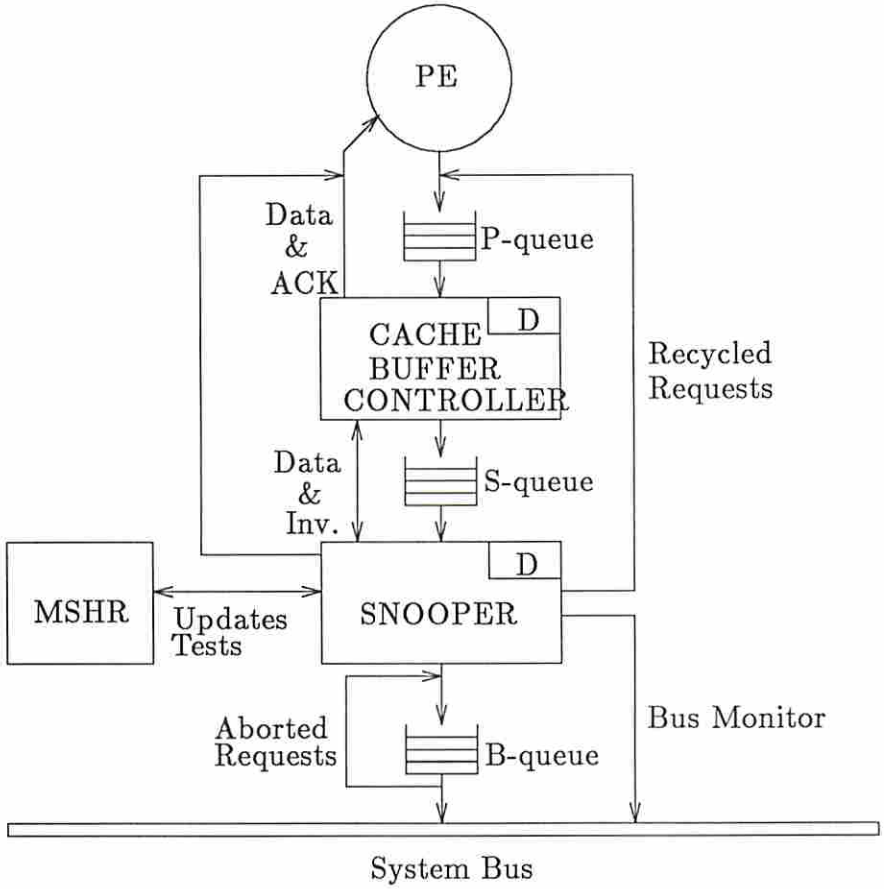


Figure 4.2: *Lockup–free cache architecture. Only one snooper and corresponding bus are shown.*

(read or write), the block address, and the destination register in the processor which will receive either the word or an acknowledgement upon miss resolution, depending whether the access is a read or a write. Each MSHR entry also contains a single flag E which indicates whether the miss is still internal or is presently being processed externally. An access is considered external as soon as it has been successfully (i.e., it has not been aborted by another cache) placed on the bus. The snooper also contains a dual tag directory to be able to properly respond to bus activities. Note that Figure 4.2 shows the logical flow of data and should not be construed to depict the actual physical interconnections within the cache.

4.4.2.1 Read hit and write hit on RW

A request taken by the cache buffer controller from the P-queue is either a read or a write access. If it is a read hit it is serviced and the processor is supplied with the requested datum. If it is a write request on a block in state **RW** it is also immediately executed and an acknowledgement is sent to the processor. The destination of a read word or an acknowledgement is packaged along with the access request.

4.4.2.2 Read miss, write miss, and write hit on RO

A request taken by the cache buffer controller from the P-queue causing a miss is placed in the S-queue to be serviced by the snooper. If the access is a write which hits in cache but the block is in state **RO** then the block is invalidated (in both directories) and the access is placed in the S-queue; from now on it is indistinguishable from a write miss. The invalidation of the tag entry in the dual directory located at the snooper bypasses the S-queue, using the direct connection between cache buffer controller and snooper. Partial–word writes are tagged as such (usually such accesses can be identified by their word-boundary address offset) and are correctly integrated into the block (word) upon miss resolution.

4.4.2.3 Snooper activity

The snooper services the S-queue. A request taken from the S-queue is first compared to the entries of the MSHR registers. If a match occurs then a miss is already pending for the block. In this case the request is recycled to the P-queue. If no MSHR entry exists for the block then one is generated and the request is placed in the B-queue. The E flag of the MSHR entry is reset to indicate that the miss is still internal (to the processor node). A reset E flag indicates that accesses and invalidations of the block made by other processors and placed on the bus need not be aborted.

The B-queue is serviced whenever the bus becomes available to the processor node. B-queue requests are placed on the bus as block read requests or **RW** block read requests. Such requests may be aborted by other snoopers by raising the busy line. In this case the aborted request is recycled to the beginning of the B-queue. A successful request is purged from the B-queue and causes the E flag to be set in the appropriate MSHR entry. Until the miss has been completely resolved, the snooper will abort all access requests to the block by other caches.

The snooper monitors the bus for three types of activities:

1. Returned blocks. If the original miss was a read then the relevant word is extracted from the block by the snooper and directly passed to the processor. The block is then transferred to the cache buffer controller where it is allocated. Both directories are updated to indicate that the block is in state **RO**. If the original miss was a write then the relevant word (or partial word) is "integrated" into the newly arrived block. An acknowledgement is sent to the processor, indicating that the write has taken place. The block is passed to the cache controller where it is allocated. Both directories are updated to indicate that the block is present in state **RW**. After a returned miss has been allocated to the cache, its MSHR entry is cleared.

2. Conflicting requests. The addresses of block read requests and **RW** block read requests appearing on the bus are compared both to the MSHR entries

and the dual directory. In case there is a matching MSHR entry *and* its E flag is set the access is aborted by raising the busy line.

3. Non-conflicting requests. Bus requests which match any MSHR with reset E flags are ignored. (A MSHR entry with a reset E flag indicates that the block *must* be presently invalid in the cache and hence no action needs to be taken.) If no MSHR entry exists for an access, then the dual directory is checked. If the accessed block is indicated as present then:

   - If the access is a read block request and the block is present in state **RW**, then the snooper requests the block from the cache buffer controller and places it on the bus in response. (This cache–to–cache transfer also results in a write–back of the block to main memory.) Both directories are updated to indicate that the block is now in state **RO**.

   - If the access is a RW read block request and the block is present in state **RW**, then the same activity as in the in the previous case takes place, except that (1) the block is invalidated in both directories after the transfer and (2) main memory is not updated during the transfer.

   - If the access is a **RW** read block request and the block is present in state **RO**, then the block is invalidated in both directories.

4.4.2.4 Block replacements

For replacement purposes, any block in state **RO** can be overwritten since a copy must also be present in main memory. If it becomes necessary to replace a block in state **RW**, care must be taken not to overlap the replacement with another cache's access to the block. This can be accomplished efficiently by using the existing busy line.

4.5 Analysis

Two models are presented here. Both models are approximate, but useful information can be derived from them. We make the following assumptions in our model:

1. A processor makes d memory references per instruction.

2. The distance between references with dependencies in the reference string of a process is fixed and is equal to l. Hence, after a miss, up to l references can be made before the processor blocks and has to wait for the miss to be resolved.

3. For each access the probability of a hit is h and the probability of a miss is $(1-h)$. Successive accesses are independent. Therefore, the number of references between two consecutive misses is geometrically distributed with mean $1/(1-h)$. (Figure 4.3 illustrates the concept.)

4. The memory access time is constant and equal to T_m.

5. The time to execute an instruction if all accesses hit in the cache is constant and equal to $t_{i,0}$. We associate a time of $t_{i,0}/d$ with each reference.

6. Effects of synchronization and of TLB misses are neglected.

There are several approximations in the model. First of all, in a practical system the dependency distance is usually variable and the memory access time is random because of memory conflicts. These two approximations were made here to facilitate the solution of the model. Second, successive accesses to the cache are correlated; however, the hypothesis of independent accesses is often made in cache models (see for example [49]) and is as good as any other hypothesis in the absence of real program traces. Overall, we feel that the models include the most important parameters affecting the performance of the lockup-free caches and should give indications as to system configurations for which the complexity of lockup-free caches is warranted.

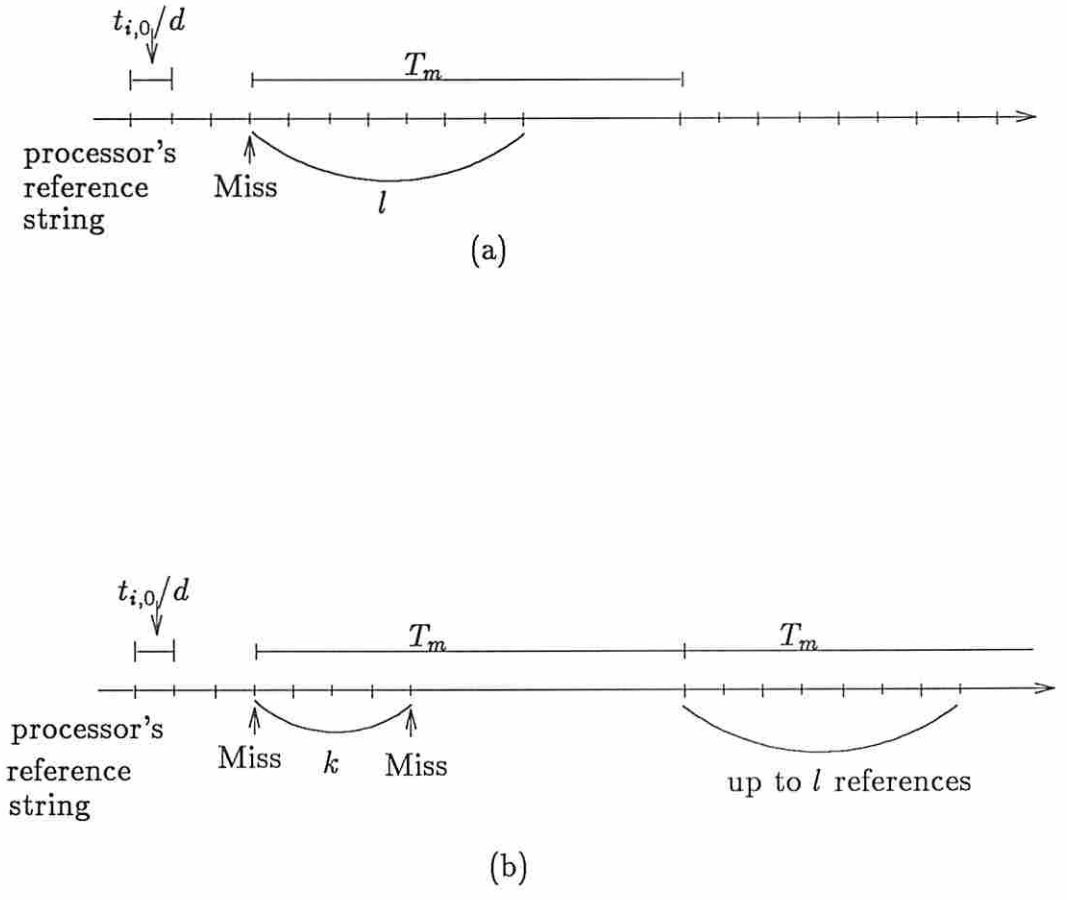


Figure 4.3: *Timing of a lockup-free cache with one buffer. (a) No miss occurs in the l references following the first miss; maximum overlap is achieved. (b) A miss occurs. Only $k < l$ references can be overlapped.*

4.5.1 Model 1: One MSHR register

It is assumed that only one MSHR register exists in the cache. A single miss does not cause the cache to lock immediately. The cache will block either if a second miss occurs, or if a dependency with the reference of the first miss prohibits any further prefetching. We have to consider two different cases. In the first case, the memory access time is larger than the time during which the processor can continue prefetching without encountering a dependency with a previous miss. That is, $\frac{t_{i,0}l}{d} < T_m$. In the second case, we assume the opposite, that is $\frac{t_{i,0}l}{d} \geq T_m$.

Case 1: When a miss is encountered, it is immediately forwarded to the memory. The number of references which can be overlapped while the miss is being resolved is $\bar{a}$. This number is governed by the probability that a miss occurs during the next l accesses before the processor blocks due to a dependency with an access that missed (Figure 4.3). Hence $\bar{a}$ is given by:

$$\bar{a} = (1-h) + 2h(1-h) + 3h^2(1-h) + \cdots$$

$$+(l-1)h^{l-2}(1-h) + lh^{l-1}(1-h) + lh^l$$

which can be reduced to:

$$\bar{a} = \frac{1-h^l}{1-h}$$

If T_N is the time to execute N instructions then

$$T_N = N(t_{i,0} + (1-h)dT_m) - N\frac{1-h^l}{1-h}(1-h)t_{i,0}$$

The time to execute one instruction is:

$$t_{i,0}h^l + (1-h)dT_m$$

Hence the slowdown factor is:

$$\frac{1}{h^l + (1-h)dT_m^0}$$

Case 2: In this case the amount of overlap of hits is only limited by the probability that a second miss occurs before the first miss is resolved. This

case may result if the code is restructured by the compiler such that the distance between dependencies which cause the processor to block is very large. Let N be the number of references between two successive misses. $\bar{N} = 1/(1-h)$ and the average time to execute between two misses becomes

$$T_N = \frac{t_{i,0}}{d} \sum_{k=0}^{\infty} kP[N=k] = \frac{t_{i,0}}{d} \sum_{k=1}^{\infty} P[N \geq k]$$

$$= \frac{t_{i,0}}{d} \left[\sum_{k=1}^{dT_m^0} 1 + h^{dT_m^0} + h^{dT_m^0+1} + \cdots \right]$$

$$= T_m + \frac{t_{i,0}}{d} \frac{h^{dT_m^0}}{1-h}$$

This result comes from the fact that if the inter-miss distance N is less than dT_m^0, the cache blocks the processor for a time $T_m - \frac{Nt_{i,0}}{d}$. The average time per instruction is given by:

$$\frac{dT_m + t_{i,0}\frac{h^{dT_m^0}}{1-h}}{\frac{1}{1-h}}$$

The slowdown factor for this case is:

$$\frac{1}{h^{dT_m^0} + (1-h)dT_m^0}$$

4.5.2 Model 2: Infinite Number of MSHR Registers

In this model, it is assumed that the number of MSHR registers is infinite. As for the previous model, we consider two cases.

Case 1: In this case we assume $t_{i,0}l/d < T_m$. After the first miss, at reference $1/(1-h)$, l references can be generated by the processor while the first miss is resolved by the memory system in time T_m. Therefore, a processing time of $t_{i,0}l/d$ can be overlapped with the first miss. If any one of these l references causes a miss, it can be serviced immediately because there are an infinite number of buffers. The misses occurring for the l references do not cause additional blocking due to dependencies because the processor has to wait until the first

miss is resolved before initiating new references. When the first miss is completed, the processor can continue execution, and because of the geometric distribution assumption, the next miss occurs $1/(1-h)$ references later, on the average. The sequence of events after the first miss is repeated.

Therefore, in a time

$$\frac{t_{i,0}}{d}\frac{1}{1-h}+T_m$$

a number of $\frac{1}{1-h}+l$ references are performed, corresponding to

$$\left(\frac{1}{1-h}+l\right)\frac{1}{d}$$

instructions.

The average time per instruction is

$$\frac{t_{i,0}+T_m d(1-h)}{1+l(1-h)}$$

and the slowdown factor is

$$\frac{1+l(1-h)}{1+T_m^0 d(1-h)}$$

Case 2: In this case we assume that $t_{i,0}l/d \geq T_m$. When a reference has a dependency with a previous access that missed, the miss has had the time to complete and therefore, the dependencies do not block the processor. We have achieved total overlap of miss handling and the slowdown factor reaches its maximum value of 1.

4.6 Discussion

4.6.1 Performance interpretations

Figure 4.4 shows the ratios of MIPS rates of a system with one MSHR buffer per cache and of a system with locking caches. The improvement is greatest for low hit rates and memory access times in the range of $T_m^0 < l/d$. This is due to the fact that if $T_m^0 > l/d$, most misses will cause some blocking, because before

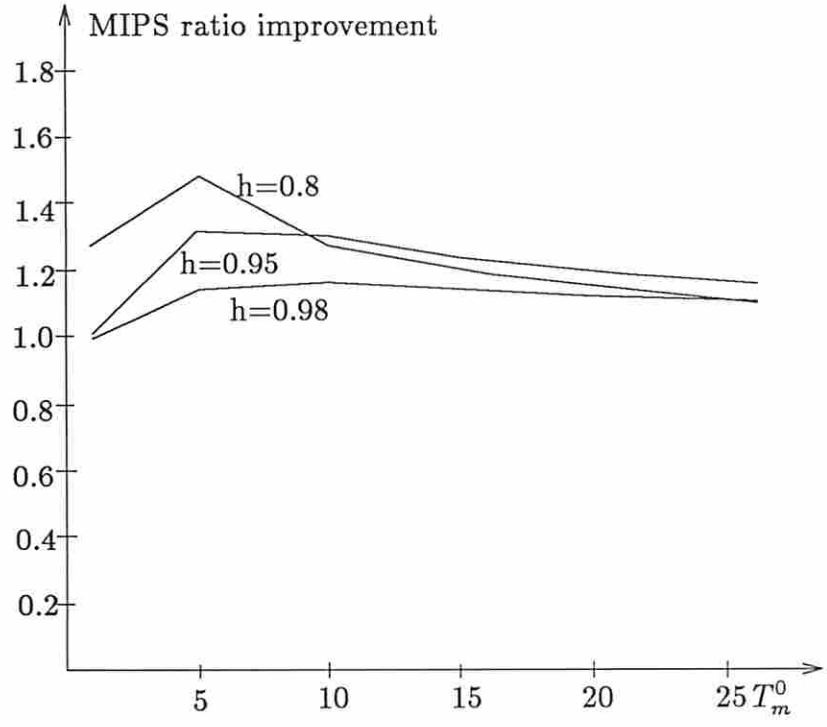


Figure 4.4: *MIPS ratio improvement with one miss buffer,* $d = 1.5$, $l = 10$.

a miss can be resolved a dependency will occur with the reference that misses. If $T_m^0 < l/d$, however, the cache will only lock if a second miss occurs during the service of the first miss; some misses will be totally overlapped with other references and the amount of overlap also increases with T_m^0.

For larger hit rates, such as h=0.98, the improvement due to the lockup-free cache with only one MSHR is low because misses are rare and the only savings per miss, with respect to a locking cache, are l references. These results confirm Kroft's results for the average access time as a function of the number of MSHR registers. Kroft's results (derived from a prototype) indicate very poor performance for a cache with a single MSHR register and very good performance for a cache with up to four MSHR registers. Kroft further states that very little is to be gained by using more than four registers.

Figure 4.5 shows the ratio of MIPS rates of a system with an infinite number of MSHR buffers and a system with locking caches for different hit rates, as a function of T_m^0. The improvement of performance is highest for a low hit rate of h=0.8. This is to be expected, since a high miss ratio results in multiple misses in a streak of l consecutive references and these misses can be overlapped. In the ideal case, for example, l misses occur sequentially, until the system blocks due to a dependency with the first miss (we assume $l/d < T_m^0$). The misses are resolved one after another and the total penalty paid for the sequence of l misses is only T_m^0. Hence, the system benefits from frequent misses.

For $l/d > T_m^0$, the system with an infinite number of buffers operates at maximum speed, with a slowdown factor of 1. In this case no miss ever causes a penalty, since the system does not block on multiple misses and all misses are always resolved before a dependency can occur. For higher hit rates the probability of overlapping misses decreases and the performance ratio degenerates to the case of the cache with only one MSHR buffer. The number of buffers used, on average, is an interesting parameter. It is possible to estimate the average number of busy buffers in the cache as follows. Let t_i be the average time per instruction. The average time to execute N instructions is Nt_i and a total of

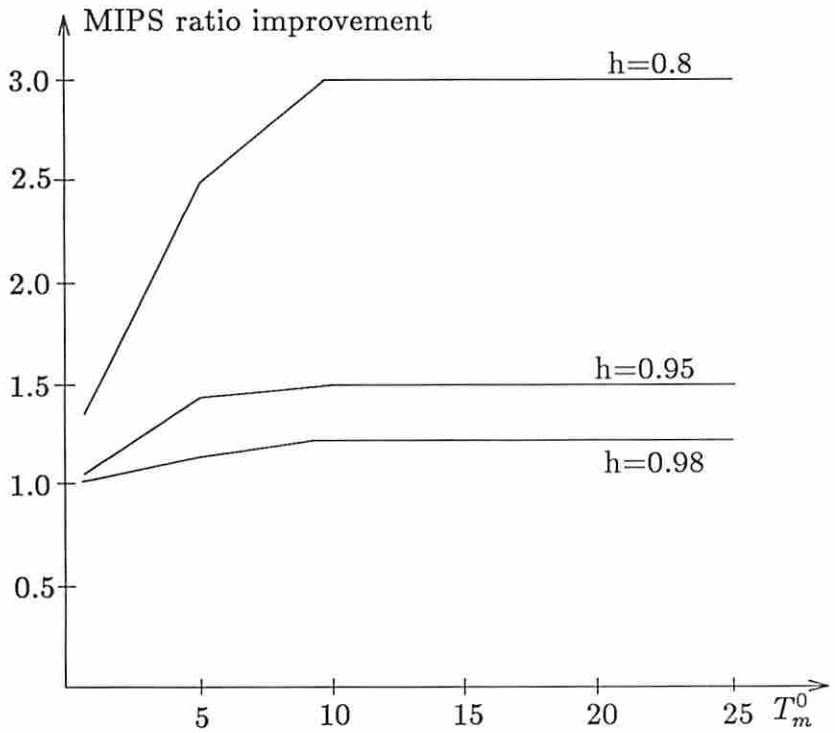


Figure 4.5: *MIPS ratio improvement with an infinite number of miss buffers,* $d = 1.5$, $l = 10$.

$Nd(1-h)$ misses must be processed, requiring a total service time of $Nd(1-h)T_m$ from the miss buffers. Therefore the average number of busy buffers is

$$\frac{d(1-h)T_m}{t_i} = \frac{d(1-h)T_m^0}{1+d(1-h)T_m^0}[1+l(1-h)] \quad (case\ 1)$$

$$or \ = d(1-h)T_m^0 \quad (case\ 2)$$

This value is upper-bounded by $1+l(1-h)$. For all of the examples of Figure 4.5, the average number of busy buffers is less than 2.75.

4.6.2 Consequences

For an otherwise efficient lockup-free cache to be of consequential benefit, multiple MSHR registers must be implemented. Only in the case when the hit rate is low, and $T_m^0 < l/d$ does a single MSHR buffer offer worthwhile improvement.

A cache with a *number* of MSHR buffers can, however, be very useful. Such a cache can offer substantial improvement over a locking cache when the hit rate is not very high. This fact can benefit systems in three particular circumstances.

1. Any system with a low hit rate benefits from lockup-free caches consistently.

2. Context switches always cause very low transitory hit rates. A lockup-free cache can help "smooth" the performance dip in the system behavior after such context switches.

3. A cache which usually exhibits a good hit rate, may be sensitive to particular "pathological" workloads which can lower the hit rate for particular applications, or, especially, for operating system calls [58]. As in the case of context switches, the cache can adapt to such workload changes if it is lockup-free.

A system with lockup-free caches, will exhibit better performance if misses are clustered together than if they are homogeneously distributed. This

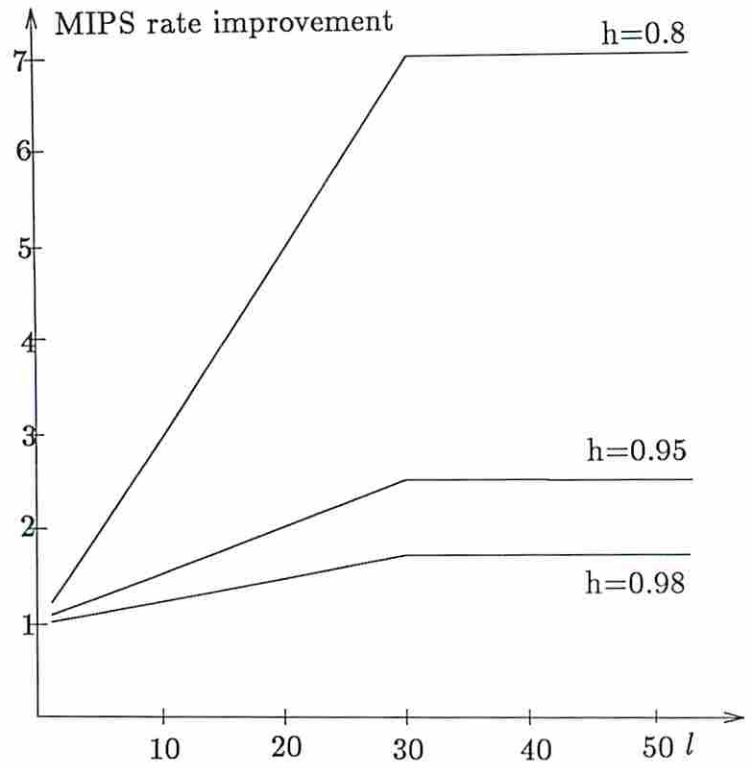


Figure 4.6: *MIPS ratio improvement as a function of* l, $T_m^0 = 20$, $d = 1.5$.

characteristic can be taken advantage of if the compiler can generate load instructions for data to be used in the future, ahead of time. In this case the "blind" prefetching associated with larger block sizes, has been replaced with selective "smart" prefetching under compiler control.

4.6.3 Dependency effects

Figure 4.6 shows the MIPS ratio of the infinite buffer system and the locking cache system as a function of l. For this case, $T_m^0 = 20$ and $d = 1.5$. As is to be expected the performance ratio increases linearly until $l/d = T_m^0$ up to a point where the lockup-free cache system operates at peak speed and remains constant. In the case of a hit rate $h = 0.8$, the maximum performance improvement over the locking cache system is 700%. Since the inter-dependency

distance l affects the performance of the system greatly, techniques on how to increase l are important.

(1) The compiler can attempt to increase l within the instruction stream by reordering instructions in a more favorable way.

(2) All load instructions should be non-blocking; as they are implemented in the IBM RT processor [29].

(3) The compiler can generate special instructions which explicitly load data into cache, long before they are needed. Such non-blocking load operations enable the compiler to control selective prefetching of data. These loads should be generated in bursts.

(4) For instruction access misses not to cause blocking, several consecutive instructions can be prefetched ahead of instruction decode time. Only branch instructions will cause blocking in this case. If branches are delayed as in RISC processors, l can be increased.

(5) Some architectures can naturally exhibit a high value for l. For example, if a vector processor is attached to the system, long strings of vector register load instructions are likely to be executed frequently.

4.7 Conclusion

In this Chapter we have shown how a multiprocessor can be configured with lockup-free caches. Vital to such a system are three key concepts:

1. The correctness of the system.
2. The efficiency of the interconnect.
3. The efficiency of the cache architecture.

We have shown that the processors of a multiprocessor system may resolve multiple misses concurrently if the system is weakly ordered. Weakly ordered multiprocessors require that shared writable data are modified exclusively from within critical sections. This restriction can be enforced by the compiler. Lockup–free cache multiprocessors can utilize most existing cache coherence pro-

tocols without major modifications. The effectiveness of such an implementation was investigated by means of an analytical model.

Overall, we believe that bus-based multiprocessors with lockup-free caches are both viable and useful. One of the most interesting features of such a system is the adaptability to the cache miss rate it exhibits. When hit rates are high, the improvement due to overlapping misses is low. However, when the hit rate declines, the efficiency of the lockup-free caches improves rapidly. This characteristic makes lockup-free caches a particularly appealing feature for systems with a large variety of types of workloads, and hence varying hit rates.

CONCLUSION

Traditionally the concept of memory coherence has been used to address most issues relating to "correctness" in parallel processing systems. This approach usually leads to either *ad hoc* engineering solutions or to a simplification of the issues at hand. For example, in systems with complex processors often a "safety first" approach is used by flushing all buffers, sometimes including caches, and by hardware locking relevant devices before any "dangerous" activities such as synchronization are attempted. Such weakly ordered systems often "stabilize" before every synchronization or communication whether it is necessary or not. Other systems are designed around serializing devices such a buses and tables, in effect intentionally creating bottlenecks.

In this thesis new concepts and definitions have been introduced which can be used to view and examine any shared memory multiprocessor in a different and more efficient way. Using these concepts, three concurrency models were defined and conditions under which systems conform to these concurrency models were derived. The concepts and definitions liberate the designer from having to examine exhaustively every parallel state contingency of a multiprocessor to convince him– or herself that the design is indeed "correct."

The usefulness of these concepts was demonstrated in Chapters three and four where they were used to outline architectures with features which are traditionally difficult to implement in multiprocessors. Namely, a cache-based system using an efficient invalidation-based protocol *without* the need for a single

invalidation bus; and a cache-based system with lockup-free caches which allow one or several cache misses to be pending at the same time.

We are hopeful that the paradigms and tools offered in this thesis can be utilized effectively in the hands of experienced designers of multiprocessors who wish to concentrate on efficient implementations rather than worry about the potential side-effects of an implementation.

Bibliography

[1] Agarwal, A., Sites, R.L., and Horowitz, M., "ATUM: A New Technique for Capturing Address Traces using Microcode," *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 119–127, 1986.

[2] Agarwal, A., Simoni, R., Hennessy, J., and Horowitz, M., "An Evaluation of Directory Schemes for Cache Coherence," *Proceedings of the 15th Annual International Symposium on Computer Architecture,* pp. 280–289, 1988.

[3] Andrews, G.R. and Schneider, F. B., "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, Vol 15, No. 1, pp. 3–43, March 1983.

[4] Archibald, J. and Baer, J.–L., "An Economical Solution to the Cache Coherence Problem," *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pp.355–362, 1984.

[5] Archibald, J. and Baer, J.–L., "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems*, Vol. 4, No. 4, pp. 273–298, November 1986.

[6] Baer, J.–L. and Girault, C., "Cache Coherence in MIMD Systems: A Petri Net Model for a Minimal State Solution," *Technical Report 87-04-01*, Department of Computer Science, University of Washington, April 4, 1987.

[7] Baer, J.–L. and Wang, W.–H., "Architectural Choices for Multilevel Cache Hierarchies," *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 258–261, 1987.

[8] Baer, J.–L. and Wang, W.–H., "On the Inclusion Properties for Multi-Level Cache Hierarchies," *Proceedings of the 15th International Symposium on Computer Architecture*, pp. 73–80, June 1988.

[9] Baudet, G.M., "The Design and Analysis of Algorithms for Asynchronous Multiprocessors," *Ph.D. Thesis, Department of Computer Science*, Carnegie-Mellon University, August 1978.

[10] Bean, B.M. *et al.*, "Bias Filter Memory for Filtering out Unnecessary Interrogations of Cache Directories in a Multiprocessor System," *U.S. patent 4,142,234*, February 27, 1979.

[11] Bitar, P. and Despain, A., "Multiprocessor Cache Synchronization: Issues, Innovations, Evolution," *Proceedings of the 13th International Symposium on Computer Architecture*, 1986.

[12] Censier, L.M. and Feautrier, P., "A New Solution to Coherence Problems in Multicache Systems," *IEEE Transactions on Computers*, Vol. C-27, No. 12, pp. 1112–1118, December 1978.

[13] Collier, W.W., "Architectures for Systems of Parallel Processes," *Technical Report TR 00.3253, IBM Corporation, Poughkeepsie*, N.Y., January 1984.

[14] Collier, W.W., "Reasoning about Parallel Architectures," *to be published by Prentice-Hall*, 1990.

[15] Conners, *et al.*, "The IBM 3033: An Inside Look," *Datamation*, pp. 198–218, May 1979.

[16] Crowther, W., Goodhue, J., Starr, E., Thomas, R., Milliken, W., and Blackadar, T., "Performance Measurements on the 128–Node Butterfly Parallel Processor," *Proceedings of the 1985 International Conference on Parallel Processing*, 1985. Blackadar, T., "Performance Measurements on the 12

[17] Dijkstra, E.W., "Solution of a Problem in Concurrent Programming Control," *Communications of the ACM*, Vol. 8, No. 9, pp. 569, September 1965.

[18] Dubois, M. and Briggs, F., "Effects of Cache Coherency in Multiprocessors," *IEEE Transactions on Computers*, Vol. C–31, No. 11, pp. 1083–1099, November 1982.

[19] Dubois, M., Scheurich, C., and Briggs, F., "Memory Access Buffering in Multiprocessors," *Proceedings of the 13th Annual International Symposium on Computer Architecture,* pp. 434–442, 1986.

[20] Dubois, M., "Effects of Invalidations on the Hit Ratio of Cache–Based Multiprocessors," *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 255–261, 1987.

[21] Dubois, M. and Wang, J.–C., "Shared Data Contention in a Cache Coherence Protocol," *Proceedings of the 1988 International Conference on Parallel Processing*, pp. 146–155, 1988.

[22] Dubois, M., Scheurich, C., and Briggs, F., "Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE Computer*, Vol. 21, No. 2, pp. 9–21, February 1988.

[23] Eggers, S. and Katz, R., "A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation," *Proceedings of the 15th Annual International Symposium on Computer Architecture,* pp. 373–382, 1988.

[24] Einstein, A., *et al.*, "The Principle of Relativity Theory," *Dover Publications Inc.*, New York, 1923.

[25] Frank, S.J., "Synapse Tightly Coupled Multiprocessor—A New Approach that Solves Old Problems," *Proceedings of the National Computer Conference*, 1984.

[26] Fuller, S.H., Swan, R., and Wulf, W.A., "The Instrumentation of C.mmp: A Multi–miniprocessor," *IEEE Compcon*, 1973.

[27] Fuller, S.H. and Harbison, S.P., "The C.mmp Multiprocessor," *Technical Report*, Carnegie–Mellon University, Computer Science Department, 1978.

[28] Gajski, D. *et al.*, "CEDAR: A Large Scale Multiprocessor," *ACM Sigarch Computer Architecture News*, March 1983.

[29] Gimarc, C., and Milutinovic, V., "A Survey of RISC Processors and Computers of the Mid–1980's," *IEEE Computer*, September 1987.

[30] Goodman, J.R., "Using Cache Memory to Reduce Processor–Memory Traffic," *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pp. 124–131, 1983.

[31] Gottlieb, A., Grishman, R., Kruskal, C., Mc Auliffe, K., Rudolph, L., and Snir, M., "The NYU Ultracomputer—Designing a MIMD, Shared Memory Parallel Machine," *IEEE Transactions on Computers*, Vol. C-32, No. 2, pp. 175–189, February 1983.

[32] Hill, M. *et al.*, "Design Decisions in Spur," *IEEE Computer*, November 1986.

[33] Hwang, K. and Briggs, F., "Computer Architecture and Parallel Processing," *McGraw–Hill*, New York, 1984.

[34] Katz, R., Eggers, S., Wood, D.A., Perkins, C., and Sheldon, R.G., "Implementing a Cache Consistency Protocol," *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pp. 276–283, 1985.

[35] Knuth, D.E., "Additional Comments on a Problem in Concurrent Programming Control," *Communications of the ACM*, Vol. 9, No. 5, May 1966.

[36] Kogge, P.M., "The Architecture of Pipelined Computers," *Mc Graw–Hill*, 1981.

[37] Kroft, D., "Lockup–free Instruction Fetch/Prefetch Cache Organization," *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pp. 81–85, June 1981.

[38] Kung, H.T., "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors," *Algorithms and Complexity: New Directions and Recent Results*, J.F. Traub Ed., New York, Academic Press, 1976.

[39] Lamport, L., "A New Solution of Dijkstra's Concurrent Programming Problem," *Communications of the ACM*, Vol. 17, No. 8, pp. 453–455, August 1974.

[40] Lamport, L., "Proving the Correctness of Multiprocess Programs," *IEEE Transactions on Software Engineering*, Vol. SE–3, No. 2, March 1977.

[41] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, Vol. 21, No. 7, July 1978.

[42] Lamport, L., "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, Vol. C–28, No. 9, pp. 690–691, September 1979.

[43] Lamport, L., "The Mutual Exclusion Problem: Part I—A Theory of Interprocess Communication," *Journal of the ACM*, Vol. 33, No. 2, pp. 312–326, April 1986.

[44] Lamport, L., "The Mutual Exclusion Problem: Part II—Statement and Solution," *Journal of the ACM*, Vol 33, No. 2, pp. 327–348, April 1986.

[45] Lee, R., Yew, P.–C., and Lawrie, D.H., "Multiprocessor Cache Design Considerations," *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pp. 253–262, June 1987.

[46] Lovett, T. and Thakkar, S., "The Symmetry Multiprocessor System," *Proceedings of the 1988 International Conference on Parallel Processing*, pp. 303–310, 1988.

[47] McCreight, E., "The Dragon Computer System An Early Overview," Xerox Corporation, September 1984.

[48] Papamarcos, M. and Patel, J., "A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories," *Proceedings of the 11th International Symposium on Computer Architecture*, pp. 348–354, 1984.

[49] Patel, J., "Analysis of Multiprocessor with Private Cache Memories," *IEEE Transactions on Computers*, Vol. C-31, No.4, April 1982.

[50] Pfister, G.F., Brantley, W.C., George, D.A., Harvey, S.L., Kleinfelder, W.J., McAuliffe, K.P., Melton, E.A., Norton, V.A., and Weiss, J.,

"The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985.

[51] Rudolph, L. and Segall, Z, "Dynamic Decentralized Cache Schemes for MIMD Parallel Processors," *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pp. 340–347, 1984.

[52] Satyanarayanan, M., "Multiprocessors: A Comparative Study", Prentice–Hall, Englewood Cliffs, N.J., 1980.

[53] Scheurich, C. and Dubois, M., "Correct Memory Operation of Cache–based Multiprocessors," *Proceedings of the 14th International Symposium on Computer Architecture*, pp. 234–243, June 1987.

[54] Scheurich, C. and Dubois, M., "Concurrent Miss Resolution in Multiprocessor Caches," *Proceedings of the 1988 International Conference on Parallel Processing*, 1988.

[55] Scheurich, C. and Dubois, M., "The Design of a Lockup-Free Cache for High-Performance Multiprocessors," *Proceedings of Supercomputing'88*, pp. 352–359, 1988.

[56] Sequent Computer Systems, "Symmetry Technical Summary," Sequent Computer Inc., Oregon, 1987.

[57] Shasha, D. and Snir, M., "Efficient and Correct Execution of Parallel Programs that Share Memory," *ACM Transactions on Programming Languages and Systems*, Vol. 10, No. 2, pp. 282–312, April 1988.

[58] Smith, A.J., "Cache Evaluation and the Impact of Workload Choices," *Proceedings of the 12th International Symposium on Computer Architecture*, 1985.

[59] Smith, A.J., "Cache Memories," *Computing Surveys*, Vol. 14, No. 3, September 1982.

[60] Smith, B.J., "A Pipelined Shared Resources MIMD Computer," *Proceedings of the 1978 International Conference on Parallel Processing*, 1978, pp. 6–8.

[61] Smith, A.J., "Line (Block) Size Choice for CPU Cache Memories," *IEEE Transactions on Computers*, Vol. C-36, No. 9, September 1987.

[62] Spirn, J.R., "Program Behavior: Models and Measurements," *Elsevier, Operating and Programming Systems Series*, 1977.

[63] Tang, C.K., "Cache System Design in the Tightly Coupled Multiprocessor System," *Proceedings of the National Computer Conference*, Vol. 45, pp. 749–753, 1976.

[64] Tucker, S.G., "The IBM 3090 system: An Overview," *IBM Systems Journal*, pp. 4–19, Vol. 25, No. 1, 1986.

[65] Vitanzi, P.M. and Auerbach, B., "Atomic Shared Register Access by Asynchronous Hardware," *Conference on the Foundations of Computer Science*, 1986.

[66] Wilson, A.W., "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," *Proceedings of the 14th International Symposium on Computer Architecture*, pp. 244–252, June 1987.

[67] Wolfe, M.J., "Techniques for Improving the Inherent Parallelism in Programs," *Technical Report R–78–929, Department of Computer Science*, University of Illinois at Urbana–Champaign, July 1978.

[68] Yang, Q. and Bhuyan, N., "A Queueing Network Model for a Cache Coherence Protocol on Multiple–bus Multiprocessors," *Proceedings of the 1988 International Conference on Parallel Processing*, pp. 130–137.