Master Thesis IMIT/LECS/ 2004 - 16

Architectural and Programming Support for Fine-Grain Synchronization in Shared-Memory Multiprocessors

Master of Science Thesis
In Electronic System Design

By

HARI SHANKER SHARMA
MS (System-on-Chip Design)
IMIT/KTH

Stockholm, Sweden, April 2004

Supervisor and Examiner: Assoc. Prof. Vlad Vlassov
IMIT/KTH
vlad@it.kth.se

Abstract

As the multiprocessors scale beyond the limits of a few tens of processors, we must look beyond the traditional methods of synchronization to minimize serialization and achieve high degrees of parallelism required to utilize large machines. Since synchronization is a major performance parameter for such a level of parallelism, efficient support for synchronization is therefore a major issue. By allowing synchronization at the level of smallest unit of memory, fine-grain synchronization achieves this goal and it has significant performance as compare to traditional coarse-grain synchronization.

It has already been proved that hardware support for fine-grain synchronization provides significant improvement in the performance over coarse-grain synchronization mechanisms like barriers. As demonstrated by the machine MIT Alewife, integrated support for fine-grain synchronization can have significant performance benefits over coarse-grain. The major goal of research is to evaluate the efficient way to support the fine-grain synchronization mechanisms in multiprocessors. The best way of approaching to this goal is based on the efficient combination of fine-grain synchronization with cache coherence protocol with the full/empty tagged shared memory (EF-memory).

We propose to design a full/empty tagged memory hierarchy with aggressive hardware support for fine-grain synchronization that is embedded in the cache coherence mechanism of a SMP or NUMA multiprocessor. It is expected that handling synchronization and coherence together can provide a more efficient platform of execution, reducing the occupancy in memory controllers and the network bandwidth consumed by the protocol messages. Our objective is to improve the performance of the full/empty synchronization mechanism such as implemented in the MIT Alewife machine, by integrating a cache coherency mechanism with the full/empty synchronization. We uses the SimpleScalar simulator to simulate our propose design for the verification and performance evaluation.

Keywords: FE-bits, Pending-bits, Fine-grain Synchronization (FGS), Shared Memory, Cache Coherence Protocol

Acknowledgements

During the course work and thesis work of my master studies at Royal Institute of Technology, I have been very fortunate to work at such a great research oriented place and all the time surrounded by my colleagues who have continually offered me their moral support, encouragement, and help needed any time.

I would like to express special thank to my supervisor and examiner of my master thesis, Prof. Vladimir Vlassov, for providing me the opportunity to work under his supervision. I am grateful to him for giving his excellent supervision and having frequent discussions with me in the related areas of my work and it made me very comfortable during my entire thesis work. He has always patiently listened to my all doubts and came up with the absolute solutions.

I would also like to thank to Prof Csaba Andras Moritz (Faculty, UMASS) and Raksit Ashok for building up my knowledge in the beginning of my thesis work and it really helped to get into the subject.

I would also like to mention some faculty names, Prof. Hannu Tenhunen, Dr. Elena Dubrova and Dr. Johnny Öberg, whom I really admire in the academia field for their excellence.

My friends also deserve a special mention here. Hearty thank to Vijay Kella, Neeraj Gupta, Mayur Pal and others whom I have not mentioned here for making my stay in Stockholm very pleasant.

Last but not the least; I would like to thank to my family for their patience, understanding and moral support while my stay outside the home for long time. I dedicate my work to them.

The list of names to be thanked is never ending here, thank to all who have given a bit smile in my life at any moment.

Hari Shanker Sharma
IT University (KTH)
Stockholm, Sweden
28 April, 2004

Contents

Table of Figures

Table of Tables

1. Introduction and motivation

The last few years have seen the introduction of a number of parallel processing systems with truly impressive maximum performance [3]. The performance in parallel and distributed computing has emerged to be one of the promising developments. It has extended the activity of human capabilities in many fields, such as numeric simulation and modeling of physical phenomena and complex systems, and different form of information processing on the internet.

Continuous advancement in the technology (i.e. improvement in logic density and clock frequency) has resulted in highly capable and complex multiprocessors. More and more parallelism has been exploited at different granularity levels (instructions, threads, processes) in programs, to best utilize the increasing capability of multiprocessors. In parallel and concurrent programming, synchronization of parallel processes is an important mechanism. It ensures the true data dependency and timing constraints. True data dependency implies that consumer should read the value only after it has been produced by producer at the specific memory location.

Synchronization incurs an overhead because of a loss of parallelism and cost of synchronization itself. For a program to execute efficiently on a multiprocessor the serialization imposed by the synchronization structure of the program must be reduced as much as possible and the overhead of the synchronization operations must be small compared to real time computation. Multiprocessors have traditionally supported only coarse-grain synchronization (for example barriers and mutual locks). Barriers divide the program in several phases (production phase, consumption phase etc). The computation or thread of next phase depends on the results of earlier phase; parallelism across the phases is prevented due to barriers. Since coarse-grain synchronization is convenient to the programmer but it's not very much feasible on massive-parallel fine grained system.

It is known that fine-grain synchronization is an efficient way to enhance the performance of many applications, provided it can be implemented efficiently. In the case of data dependence, fine-grain synchronization allows the amount of data transferred from one thread to other threads in one synchronization operation to be small (for example one word or small cache block). The *MIT Alewife* architecture [1], [2], [3] is one that supports the fine-grain synchronization and shows demonstrable benefits over a coarse-grain approach. The Alewife multiprocessor [22] however, implements synchronization in a software layer (with some hardware support) above the cache coherence layer. Keeping these both layers separately and synchronize the computation incurs additional overhead.

So, the novel idea is to combine synchronization layer and coherence layer into one. This work describes Synchronization Coherence, a novel architecture where fine-grained synchronization and cache coherence are handled uniformly and efficiently. We propose a *full/empty tagged* hierarchy with aggressive hardware support for fine-grain synchronization embedded in cache coherence mechanism [26], [29]. This approach has two major advantages: (1) *Synchronization misses* are treated as cache misses and are resolved transparently. If we compare this with Alewife, where trap is fired on synchronization misses. This trap keeps polling the location until synchronization is satisfied, or context switches to another ready thread after a certain waiting period. This is very expensive task along with associated complexity for thread scheduling and it can avoid by utilizing above mentioned architecture. (2) Tighter integration between synchronization and cache coherence layers results fewer network messages, translating into lower network contention and improved performance.

There is a need to extend the source of *Simplescalar Simulator* to model the proposed cc-NUMA architecture to evaluate it performance. Evaluation of the aggressive hardware support in fine-gain synchronization is the main goal of this project. Software applications like *MICCG3D* from *SPLASH 2* can be used to evaluate and compare the performance of the proposed architecture with the existing *Alewife Architecture.*

The rest of the report is organized as follows. Chapter 2 gives a primer on the overview of synchronization semantics from both the view: programming language issues and memory operations. Chapter 3 deliberates on the description of Alewife machine and on the proposed architecture. Chapter 4 describes the integration of fine-grain synchronization *(FGS)* with snoop-based protocol. Chapter 5 presents the integration of *FGS* with directory-based protocol. Chapter 6 gives the detail of evaluation framework. Finally Chapter 7 and Chapter 8 describe the conclusion and future work.

2. Overview of Synchronization

A critical interplay of hardware and software in multiprocessors arises in supporting synchronization operations: mutual exclusion, point-to-point events and global events. There has been considerable debate over the years about how much hardware support exactly and what hardware primitive should be provided to support these synchronization operations [12]. Hardware support has the advantage of speed but the software has the advantages of low cost, flexibility and adaptability to different situations.

Synchronization in shared-memory multiprocessors ensures correctness by enforcing two conditions: read-after-write data dependency and mutual exclusion. Read-after-write data dependency is a contract between a producer and a consumer of shared data. It ensures that a consumer reads a value only after it has been written by a producer. Mutual exclusion enforces atomicity. When a data object is accessed by multiple threads, mutual exclusion allows accesses of specific thread to proceed without intervening accesses by other threads.

A coarse-grain solution to enforcing read-after-write data dependency is barrier synchronization. Barriers are typically used in programs involving several phase of computation where the values produced by one phase are required in the computation of subsequent phases. Parallelism is realized within a single phase, but between phases, a barrier is imposed which requires that all work from one phase be completed before the next phase is begun. Under the producer-consumer model, this means that all the consumers in the system must wait for all the producers at a common synchronization point. A fine-grain solution provides synchronization at the data level. Instead of waiting on all the producers, fine-grain synchronization allows a consumer to wait only for the data that it is trying to consume. Once the needed data is made available by the producer(s), the consumer is allowed to continue processing. Fine-grain synchronization provides two primary benefits over coarse-grain synchronization [32]:

- Unnecessary waiting is avoided because a consumer waits only for the data it needs.
- Global communication is eliminated because consumers communicate only with those producers upon which they depend.

The significance of the first benefit is that parallelism is not artificially limited. Barriers impose false dependencies and thus inhibit parallelism because of unnecessary waiting. The significance of the second benefit is that each fine-grain synchronization operation is much less costly than a barrier. This means that synchronizations can occur more frequently without incurring significant overhead. Following are three mechanisms to support fine-grain synchronization:

- Language-level support for the expression of fine-grain synchronization.
- Memory hardware support to compactly store synchronization state.
- Processor hardware support to operate efficiently on synchronization state.

The first mechanism of support provides the programmer with a means to express synchronization at a fine granularity resulting in increased parallelism. Another attractive consequence is simpler, more elegant code [8]. The second mechanism of support addresses the fact that an application using fine-grain synchronization will need a large synchronization name space. Providing special synchronization state can lead to an efficient implementation from the standpoint of the memory system. We refer to this benefit as *memory efficiency*. Finally, the last mechanism of support addresses the fact that synchronizations will occur frequently. Therefore, support for the manipulation of synchronization objects can reduce the number of processor cycles incurred. We refer to this benefit as *cycle efficiency*.

2.1. Programming Language Issues

It is desirable that fine-grain parallelism and synchronization be expressible at the language level. Programmer has freedom to specify which parts of a program may be executed in parallel. This does not preclude a compilation phase that converts sequential to parallel code. It is up to the system which part to execute in parallel and to handle proper synchronization. There are two ways a programmer may express parallelism in a program [20]:

2.1.1. Data level parallelism

Data –level parallelism express the application of some function to all or some elements of an aggregate data object, such as an array. Data-level parallelism is often expressed using parallel do-loops. Synchronization with in parallel do-loop can be either coarse-grain or fine-grain. For producer-consumer synchronization, coarse-grain synchronization involves placing a **barrier** at the end of the loop. Elements of the aggregate are written by threads using ordinary stores. At the end, each thread waits for all to complete. The values can then be accessed with ordinary reads. Alternatively, in the case of mutual-exclusion locks, coarse-grain synchronization will associate a lock with a large chunk of data.

Fine-grain data-level synchronization is expressed using data structure with accessors that implicitly synchronize. We call these structures J-structure and L-structure arrays. A J-structure is inspired by I-structure.

i) I-structure

This is widespread agreement that only parallelism can bring significant improvement in computing speed (several orders of magnitude faster than today's supercomputers). Functional languages have received much attention as appropriate vehicles for programming parallel machines for several reasons. They are high-level, declarative languages, insulating the programmer from architectural details. Their operational semantics in terms of rewrite rules offer plenty of exploitable parallelism, freeing the programmer from details of scheduling and synchronization of parallel activities.

Later it is realized that there are some difficulties in the treatment of data structures in functional languages and then I-structure is proposed [8]. I-structure is an alternative way to treat the data structures. We can compare the solutions of any test application using functional data structure and I-structure and performance of the structure can be evaluated on the basis of following points:

- efficiency (amount of unnecessary copying, speed of access, number of reads and writes, overheads in construction etc)
- parallelism (amount of unnecessary sequentialization)
- ease of encoding

It has been already investigated that it is very difficult to achieve all three objectives using functional data structures. Since the idea about I-structure evolved in the context of scientific computing, most of the discussion is couched in terms of *arrays*. I-structure grew out of a long-standing goal to have functional languages suitable for *general-purpose* computation, which included scientific computations and the array data-structures that are endemic to them. The term I-structure has been used for two separate concepts. One is and architectural idea, that is the implementation of synchronization mechanism in hardware. The other is a language construct, a way to express incrementally constructed data structure.

Based on experiments at MIT, it has been observed that I-structures solve some of the problems that arise with functional data structures, still there are another class of problems for which they still do not lead to efficient solutions and require the greater flexibility of I-structures.

ii) J-structure

A J-structure is a data structure for producer-consumer style synchronization inspired by I-structures. A J-structure is like an array, but each element has additional state: *full* or *empty***.** The initial state of a J-structure element is empty. A reader of an element waits until the element's state is full before returning the value. A writer of a J-structure element writes a value, sets the state to full, and release any waiting readers. An error signaled if a write is attempted on a full element. The difference between J-structure and I- structure is that, to enable efficient memory allocation and good cache performance, J-structure elements can be reset to an empty (unbounded) state.

iii) L-structure

L-structures are arrays of "*lock-able*" elements that support three operations: a locking read, a non-locking peek, and a synchronizing write. A locking read waits until an element is full before emptying it (i.e. locking it) and returning the value. A non-locking peek also waits until the element is full, but then returns the value *without* emptying the element. A synchronizing write stores the value to an empty element, and sets the location to full and release all read waiters, if any. As for J-structures, an error is signaled if the location is already full. A L-structure therefore allows mutually exclusive access to each of its elements. The synchronizing L-structure reads and writes can be used to implement M-structures. However, L-structures are different from M-structures in that they allow multiple non-locking readers, and a store to a full element signals as an error.

2.1.2. Control Parallelism

Using control parallelism, a programmer specifies that a given expression 'E' may be executed in parallel with the current thread. Synchronization between these threads is implicit and occurs when the current thread demands, or touches, the value of 'E'. The programmer does not have to explicitly specify each point in the program where a value is being touched. A touch implicitly occurs anytime a value is used in an ALU operation or as a pointer to be dereferenced, but not when a value is returned from a procedure or passed as an argument to a procedure. Storing a value into a data structure also does not touch the value.

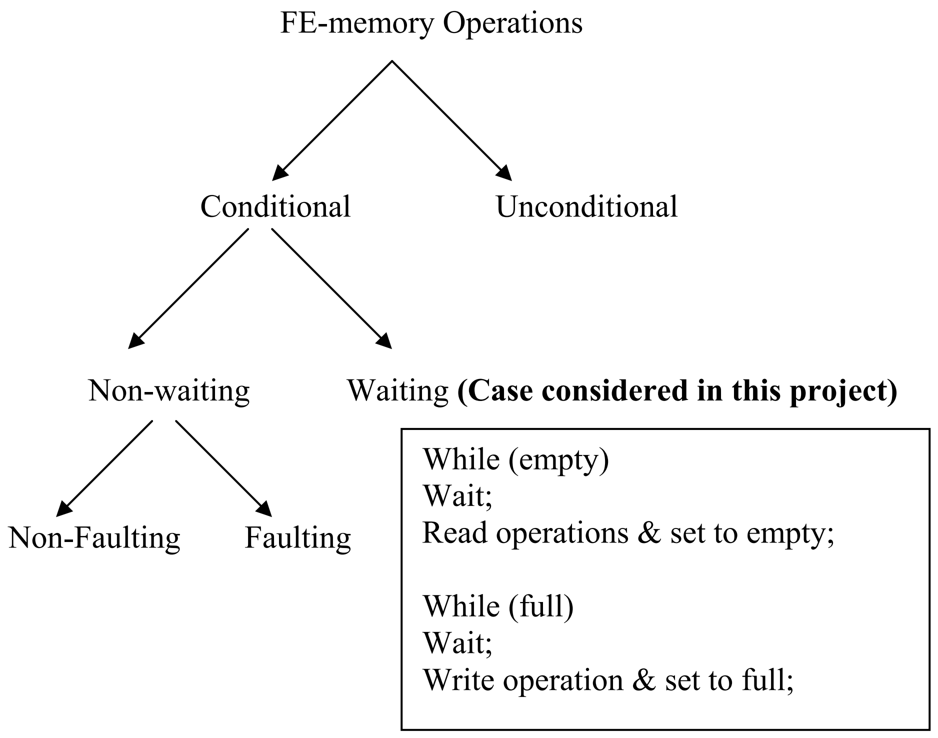
In Alewife machine, the behavior that a given expression 'E' may be executed in parallel with the current thread, is specified by wrapping *future* around an expression or statement 'E'. The keyword *future* does not necessarily cause a new runtime thread to be created, together with the consequent overhead. The system must, however, ensure that the current thread and 'E' can be executed concurrently if necessary (e.g. to avoid deadlock), when a new thread is created at runtime only for deadlock avoidance or load-balancing purposes, it is called *lazy task creation***.**

Using *future* provides a form of fine-grain synchronization because synchronization can occur between the producer and consumers of an arbitrary expression, e.g. a producer call can start executing while some of its arguments are still being computed

2.2. Semantics of synchronizing memory operations

Synchronization memory operation requires the use of tagged memory, in which each location is associated to a state bit in addition to a 32-bit value. The state bit is known as *full/empty* bit (FE-bit) and implements the semantics of synchronizing memory accesses. This state bit basically controls the behavior of synchronized loads and stores. A set FE-bit indicates that the corresponding memory reference has been written successfully by a synchronized *store* and unset FE-bit means either that memory location has never been written since it was initialized or that a synchronized *load* has read it.

A complete category of the different synchronizing memory operations is drawn in Figure 1 [29]. These instructions are introduced as an extension of the instruction set of *SPARCLE* [4], which is in turn based on *SPARC* [30]. The operations includes unconditional *load,* unconditional *store,* setting of *FE-bit* and/or combination of all these. As they do not depend upon the previous value of the state bit, *unconditional* operations always succeed.



```
While (empty)
Wait;
Read operations & set to empty;

While (full)
Wait;
Write operation & set to full;
```

Figure 1: Classification of Synchronizing Memory operations

Conditional operations depend on the value of the FE-bit to complete successfully. For instance, conditional write can only be performed if the state of FE-bit is *unset* and vice-versa for conditional read operation. Conditional memory operation can be either *waiting* or *non-waiting*. In Conditional *waiting* operation case, the operation remains pending until the state miss is resolved. This requires that the memory keep track of outstanding state misses (pending operations) in a way that is similar to keeping track of outstanding cache misses. In this project work, we are only focusing on *conditional waiting operation*.

Conditional non-waiting memory operations can be either *faulting* or *non-faulting.* Faulting operation fire a trap on a state miss and trap handler may either retry the operation immediately (spin) or switch to another context. A *non-faulting* operation does not treat a state miss as an error and so does not require the miss to be resolved. Such operation is dropped on a state miss.

All memory operations, described in figure 1 are further classified into two categories *altering* and *non-altering* operations. Altering memory operations modify the state of FE-bit after successful synchronizing event whereas non-altering memory operations do not. According to this distinction, ordinary memory operations fall into *unconditional non-altering* category.

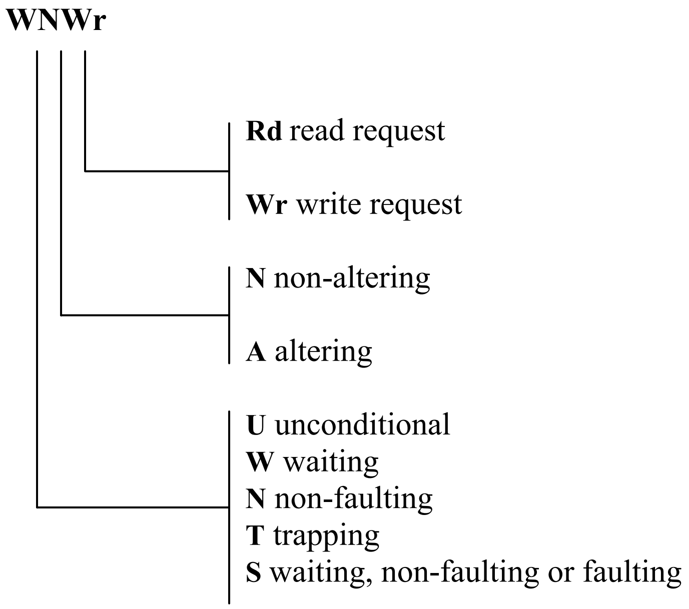


Figure 2: Notation of synchronizing memory operations

The following table describes the notation used for each variant of memory operation and its behavior in case of synchronization miss. These notations have been explained in figure 2.

Table 1: Notation of synchronized operations

Notation	**Semantics**	**Behavior on a synchronized miss**
UNRd	Unconditional non-altering read	Never miss
UNWr	Unconditional non-altering write	
UARd	Unconditional altering read	
UAWr	Unconditional altering write	
WNRd	Waiting and non-altering read from *full*	Placing on the list of pending request until resolved
WNWr	Waiting and non-altering read from *write*	
WARd	Waiting and altering read from *full*	
WAWr	Waiting and altering read from *write*	
NNRd	Non-faulting and non-altering read from *full*	Silently discarded
NNWr	Non-faulting and non-altering read from *write*	
NARd	Non-faulting and altering read from *full*	
NAWr	Non-faulting and altering read from *write*	
TNRd	Faulting and non-altering read from *full*	Signal trap
TNWr	Faulting and non-altering read from *write*	
TARd	Faulting and altering read from *full*	
TAWr	Faulting and altering read from *write*	

3. Architectural support for fine-grain Synchronization

3.1 Review of related work

3.1.1. Alewife Machine

The MIT Alewife machine [3] is a *CC – NUMA* multiprocessor with a *full/empty* tagged distributed shared memory and hardware-supported block multithreading. The machine, organized as shown in Figure 3. Memory is physically distributed over the processing nodes, which use a cost-effective mesh network for communication.

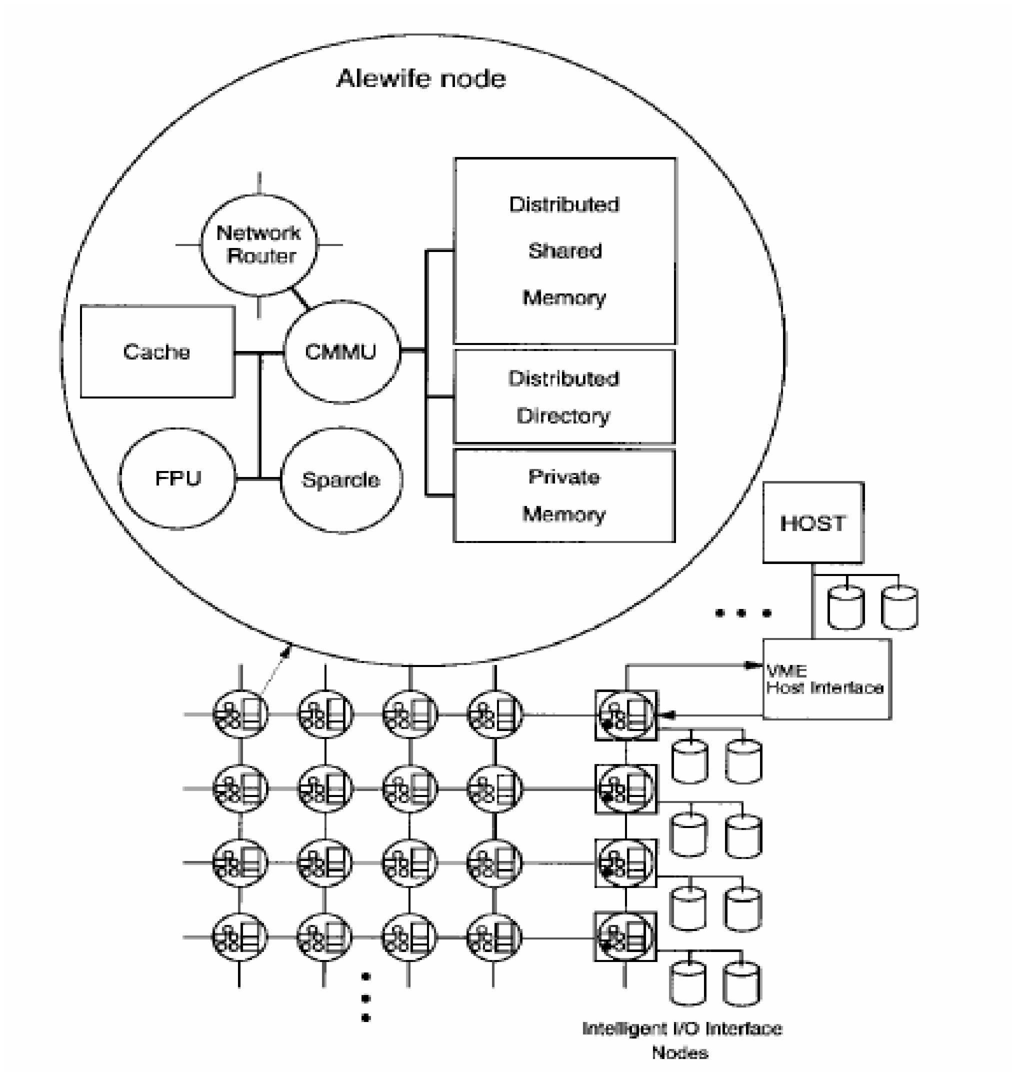


Figure 3: Alewife node, LimitLESS directory extension [9].

An Alewife node consists of a 33MHz SPARCLE processor, 64K bytes of direct-mapped cache, 4M bytes of globally-shared main memory, 2M bytes of directory (to support a 4M byte portion of shared memory), 2M bytes of unshared memory and a floating-point co-processor.

Alewife machine is internally implemented with efficient message-passing mechanism. It provides an abstraction of a global shared memory to programmers. The most relevant part of its nodes regarding coherency and synchronization protocol is the *communication and memory management unit (CMMU),* which deals with the memory request from the processor and determines whether a remote access is needed, it also manage the cache filling and replacements. Cache coherency is achieved through *LimitLESS* [9]*,* a directory based protocol. The home node is responsible for the coordination of all coherence operations for that line.

3.1.2. Hardware vs. Software approach in Alewife

In *Alewife implementation,* hardware support has been provided for the automatic detection of failure whereas actual handling of the failure is supported by the software. This technique gives an efficient CPU pipelining and register set in place, thus retaining good single thread performance

There are two ways to express parallelism in Alewife: *Data level* and *Control parallelism* [20]. While implementing *data-level parallelism*, it's necessary to synchronize at the defined granularity level. *L-structure* and *J-structure* are example of fine-grain synchronizing loads and stores. Such an operation reads or writes a data word while testing and/or setting a synchronizing condition. If operation succeeds, it doesn't take longer than a normal load or store to complete. In the event of failure, the processor fires the trap.

For the implementation of control parallelism it is necessary to know when a value produced by a future expression is being touched. This might happen anytime a value is used as an argument to an ALU operation or dereferenced as a pointer. The processor traps if the value being touched is not ready.

3.1.3. Hardware support for J-structure and L-structure in Alewife

Full/empty bits are used to represent the state of synchronized data in J-structures and L-structures. A *full/empty* bit is referenced and/or modified by a set of special load and store instructions. References and assignments to J and L-structures use the following special load, store and swap instructions depending on whether detection through traps is desired or not:

LDN	Read location
LDEN	Read location and set to empty
LDT	Read location if full, else trap
LDET	Read location and set to empty if full, else trap.
STN	Write location
STFEN	Write location and set to full
STT	Write location if empty, else trap
STFT	Write location and set to full if empty, else trap
SWAPN	Swap location and register
SWAPEN	Swap location with register and set to full
SWAPT	Swap location with register if empty, else trap
SWAPET	Swap location with register if full, else trap

In addition to possible trapping behavior, each of these instructions sets a condition code to the state of the *full/empty* bit at the time the instruction starts execution. The complier has choice to use traps or tests of this condition code. When a trap occurs, the trap handling software decides what action needs to take. These synchronization J- and L-structures provide for data-dependency and mutual exclusion, and are primitives upon which other synchronization operations can be built. Failed synchronizations are completely handled in software. In Alewife machine, failure is detected in hardware, and the trap dispatch mechanism passes control to the appropriate handler.

i) J-structure

Each *J-structure* element has been associated with a full/empty state bit. Allocating a *J-structure* is implemented by allocating a block of memory with the *full/empty* bit for each word. Resetting a J-structure element involves setting the *full/empty* bit for that element to empty. Implementing a *J-structure* read is also straightforward; it is a memory read, which fire the trap, if the full/empty bit is empty.

If the *full/empty* bit is empty, the reading thread may need to suspend execution and queue itself on a wait queue associated with the empty element. Now the question is, where this queue be stored? A possible implementation is to represent each J-structure element with two memory locations, one for the value of the element and other for a queue of waiters.

ii) L-structure

The implementation of *L-structure* is similar to that *J-structure*. The main differences are that L-structure elements are initialized to full with some initial value, and an L-structure read of an element sets the associated full/empty bit to empty and the element to the null queue. An L-structure peek, which is non-locking, is implemented in the same way as a J-structure read.

On an L-structure write, there may be multiple readers requesting mutually exclusive access to the L-structure slot. Therefore, it is wise to release only one reader instead of all readers. Here the potential problem is that if the released reader remains unscheduled for some significant length of time after being released. It is not clear what method of releasing of waiter is best, and in current implementation it releases all waiters.

3.1.4. Handling of failed synchronization in software

Due to *full/empty* bits and signaling failures via traps, successful synchronization incurs very little overhead. But in case of synchronization miss, machine fires a trap and provides enough hardware support to rapidly dispatch processor execution to a trap handler. A failed synchronization implies that the synchronizing thread has to wait until synchronization condition is satisfied. There are two fundamental ways for a thread to wait: polling and blocking [20].

Polling involves repeatedly checking the value of a memory location, returning control to the waiting thread when the location changes to the desired value. No special hardware support is needed for the *polling*. Once the trap handler has determined the memory location to poll, it can poll on behalf of the synchronizing thread by using non-trapping memory instructions, and return control to the thread when the synchronization condition is satisfied.

Blocking is more expensive because of the need to save and restore registers. Saving and restoring registers is particularly expensive in *SPARCLE* because loads take two cycles and stores three. If all registers need to be saved and restored, the cost of *blocking* can be several hundreds of cycles, more or less depending on cache hits.

In *Alewife machine*, the compiler informs the synchronization trap handler to execute in case of synchronization misses. This trap handler is basically known as a waiting algorithm. If there are other threads to execute in parallel, then appropriate waiting algorithm is to block the memory location for barrier synchronization, and to poll for a while before blocking for fine-grain producer-consumer synchronization. Since fine-grain synchronization leads to shorter wait times, this reduces the probability that a waiting thread gets blocked.

To control hardware complexity, thread scheduling is also done entirely in software. Once a thread is blocked, it is placed on a software queue associated with the failed synchronization condition. When the condition is satisfied, the thread is placed on the queue on runnable tasks at the processor on which it last run. A distributed thread scheduler that runs on all idle processors checks these queues to reschedule runnable tasks.

3.2 Proposed architecture

The main aim of this research is to design and evaluate the performance of a Full/Empty tagged memory hierarchy with the aggressive hardware support for the implementation of fine-grain synchronization embedded in a cache coherency mechanism of an *SMP* or a *NUMA* multiprocessor.

The objective here is to develop the efficient way to support the fine-grain synchronization in multiprocessor. The methodology used is to merge the fine-grain synchronization with the cache coherence protocol [26], [27], [29]. At this end, some changes are required in the existing architecture of *cc-NUMA* machines. This section is dealing with the architectural modifications that need to make to support the synchronization coherence protocol.

Assume that each memory word is associated with *full/empty bit* (FE-bit). We call such a memory *full/empty* tagged memory or simply *FE-memory*. This *FE-bit* indicates the binary state of that memory location. If this bit is set (means logical value is 1), location is *full* otherwise location is *empty* (means it is in reset state and its logical value is 0). The *full state* of the memory location can be interpreted as bound, defined and containing some meaningful value. The *empty state* of the memory location can be interpreted as unbound, undefined and containing some meaningless value. In general *FE-memory* can be considered as the composition of three logical parts [29]:

i) *Data memory* (DM) that holds the defined data.
ii) *State memory* (SM) that holds the state bit means *FE-bit.*
iii) *State miss memory* (SMM) that holds the pending access requests.

3.2.1 Architectural model

In earlier work [20], it is stated that if new synchronized *read/writes* come to such a memory location that is already *empty/full* (means *FE-bit* is reset/set), then these *read/writes* are considered as synchronization misses and interpreted as an error. To resolve this error exception is raised and it is handled differently.

In the suggested architecture, synchronized *read/write* misses are not interpreted as an error, whereas we assume that *full/empty* memory operations suspends on a synchronized *read/write* miss (by analogy to a cache miss), waiting in a memory while the miss is resolved. In this way, a queue of waiting threads (pending operations) will be maintained as a queue outstanding misses. These pending operations are stored in state miss memory. When an appropriate synchronizing operation is performed, the relevant pending requests stored in the list are resumed.

Each cache is also tagged with two pending bits with each word to provide full hardware support for completing the synchronized pending read/write memory operations.

i) ***Pr-bit***, if this bit is set, it means there is/are pending synchronized read for the corresponding word. This information is required for the synchronized write to satisfy immediately the pending synchronized read after completing the write operation into the specified memory location.

ii) ***Pw-bit*****,** if this bit is set, it means there is/are pending synchronized write for the corresponding memory location. This information is required for the synchronized read to satisfy immediately the pending synchronized write after completing all the read operations from the specified memory location.

A *FE-memory* operation might access only data (e.g. *read/write*), or data and state (e.g. read/write and set to *empty/full*), or only state (e.g. set to empty). We assume that a memory operation that accesses both, data and state is atomic. An operation is called *altering* if it sets a new state for the target location. Altering read sets the location *empty* and read the data and altering write sets the location *full* and writes the data.

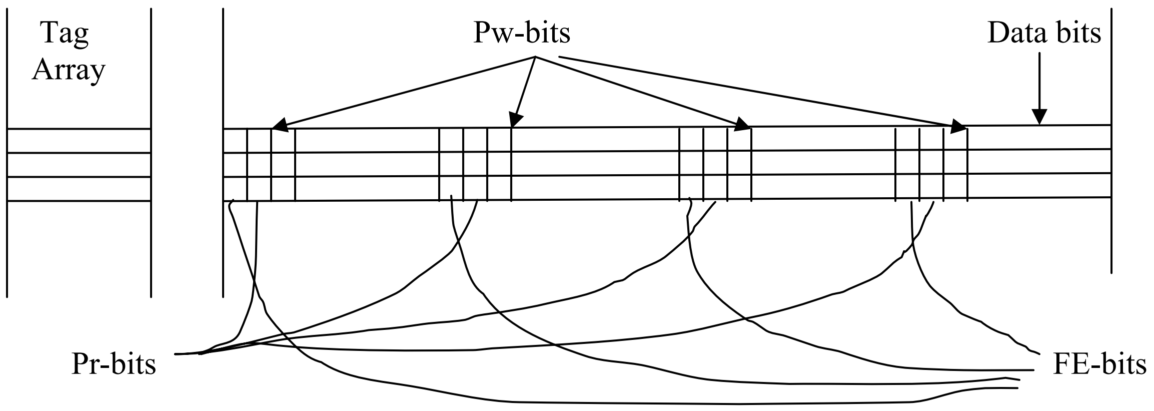


Figure 4: Architecture of Modified Cache

The Figure 4 illustrates the changes in Cache. Considering the example of 4-processor system with 32-byte memory blocks (4 words), the cache block have a storage overhead of 9% - 12 bits (4 FE-bits, 4 Pr-bits, 4 Pw-bits) extra than 256 bits data block plus tag bits.

Figure 5 illustrates a possible logical organization of a *full/empty memory* and *full/ empty cache* in a bus based shared memory multiprocessor (only one node is shown). *Empty bits* and *pending bits* can be stored together with tags in the tag-directory of the FE-cache.

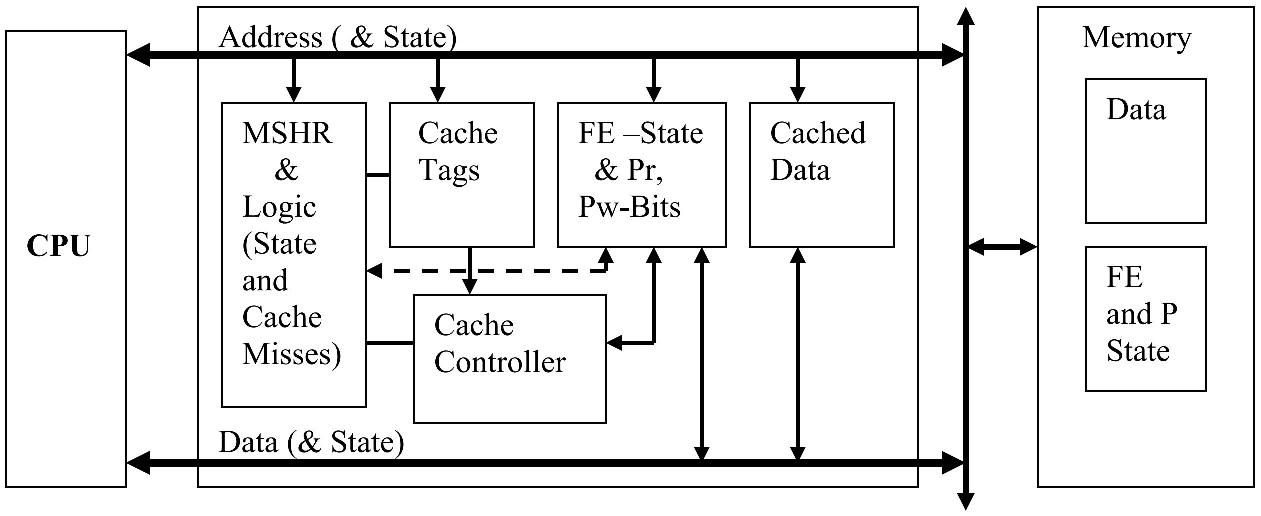


Figure 5: Organization of FE-cache and FE-memory

We assume that state misses can be treated in the same was as cache misses and the information that keeps track of outstanding misses (state and cache) is stored in *Miss State Holding Registers* (MSHR).

Some modifications have to be made to the cache architecture in case synchronization misses are to be kept in *MSHR*. More specifically, MSHR in *lockup free caches* stores the information listed in Table 2 [11], [21]. In order to store synchronization misses in these registers, two more fields have to be added containing the slot's index accessed by the operation and the specific variant of synchronized that will be performed.

Table 2: Information Stored in MSHR

Field	**Semantics**
Cache buffer address	Location where data retrieved from memory is stored
Input request address	Address of the requested data in main memory
Identification tags	Each request is marked with a unique identification label
Send-to-CPU flags	If set, returning memory data is sent to CPU
In-input stack	Data can be directly read from input stack if indicated
Number of blocks	Number of received words for a block
Valid flag	When all words have been received the register is freed
Obsolete flag	Data is not valid for cache update, so it is disposed

When a memory word is cached, its *full/empty* bit along with *pending bits* must also be cached. As a result, not only data but *full/empty* and *pending bits* must also need to keep coherent. An efficient option is to store the *full/empty bits* and *pending bits* as an extra field in cache tag, allowing the checking of synchronization state in the same step as the cache lookup. Hence the coherence protocol has two logical parts, one for data and other for synchronization bits.

This cache design coupling with fine-grain synchronization, the smallest synchronization element is a word. Since cache line is usually longer, so it may contains multiple elements, including both synchronized and ordinary data. [26] (Refer figure 6). A tag contains the *full/empty bits* and *pending bits* for all synchronized words that are stored in that line. Usually state information refers to complete cache line whereas *full/empty bit* and *pending bit* refer to single word in that cache line.

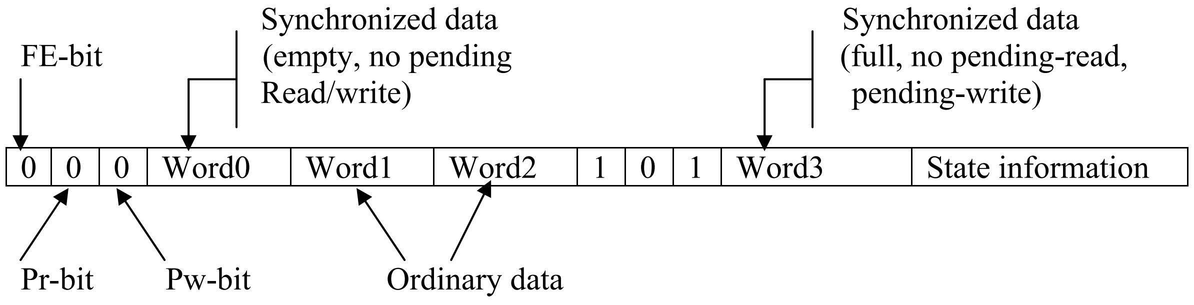


Figure 6: Cache line containing both ordinary and synchronized data.

The following table 3 explains the synchronization operation of read/writes on the synchronized memory location depending on the status of FE-bit and pending-bits.

A complete description of a cache coherence protocol includes the states, transition rules, protocol message specification and the description of a cache line organization and memory management of pending requests. The suggested architecture is based on following assumptions:

- The smallest synchronized data element is a word;
- CPU implements *out – of – order* execution of instructions;
- Each processing node has a *miss-under-miss lockup-free* cache and supporting multiple outstanding memory requests.

Table 3: Synchronized operation on synchronized data word Based on FE and P-Bits

FE-bit	**Pr-bit**	**Pw-bit**	**Synchronized operation on synchronized data word**
0	0	0	No pending Read/Write operation for the memory location, New synchronized write can be performed and set the FE-bit
0	0	1	More than one synchronized writes are pending and next synchronized write can be resumed from the pending write queue and set the FE-bit
0	1	0	Only pending-read, no pending-write, new synchronized write can be performed and set the FE-bit to resume the pending-read.
0	1	1	Pending read as well as more than one synchronized writes are pending for that location and next synchronized can be resumed from the pending write queue and set the FE-bit to resume the pending-read.
1	0	0	No pending read/write, new synchronized read can be performed to read or FE-bit can be reset to reuse of that memory location
1	0	1	Only pending-write, new synchronized read can process or FE-bit can be reset to reuse of that memory location and resume the pending-write.
1	1	0	Discarded combination
1	1	1	Discarded combination

3.3 Synchronization cache coherence protocol

In a multiprocessor system, cache memory to each processing node is used to speed up the memory operations. It is necessary to keep the cache in coherency [10] by ensuring that modifications to data that is resident in cache are seen in the rest of the node that share a copy of the data. Cache coherence can be achieved in several ways depending upon the system architecture.

[12] In *Bus-Based System*, cache coherency is implemented by *snooping mechanism,* where each cache is continuously monitoring the system bus and updating its state according to the relevant transactions seen on the bus. On the other hand *mesh network-based* system use a directory structure to ensure cache coherency. Both *snoopy* and *directory based* mechanisms can be further classifieds into *invalidate* and *update protocols.* When a cache modifies shared data, all other copies are set as invalid in case of *Invalidate Protocol,* whereas in *Update Protocol* all copies sets to new value in all cache during modification of shared data in one cache instead of making them invalid.

The performance of multiprocessor is partially limited by cache miss and node interconnects traffic [11]. Another performance issue is the overhead imposed by synchronizing data operations; this overhead is due to the fact that synchronization is implemented as a separate layer over the cache coherence protocol. If synchronization and coherence protocols are more tightly coupled by merging them into one, increased performance and reduced network traffic can be achieved.

Each memory is associated with a *full/empty bit* (FE-bit). This *FE-bit* indicated the state of memory location. If the bit is set, location is full, otherwise the location is empty. Each cache is also tagged with *pending bits* (P-bits) – if a *Pr-bit* is set, it means there is a pending synchronized read for the corresponding word and if *Pw-bit* is set, it means there is a pending synchronized write for the corresponding word. The cache controller not only match the tag bit also state bits depending on the instruction and take decision based on the state of the associated *FE-bit* and *P-bits*.

The defining feature of the Synchronization Coherence Protocol is that *synchronization misses* are treated as a *cache misses* in the individual nodes. It thus kept in the Miss Information Holding Registers of a remote node to be subsequently resolved by explicitly messages from the home node (directory). The home nodes contains Synchronization Miss Buffer (SMB), that holds the information regarding which node have pending synchronized read/write for a given word in case of *directory-based protocol*..

In order to evaluate the performance improvement of this proposed architecture with respect to existing architecture, appropriate workloads must be tested on the machine. We must find the suitable application that show the result in meaningful way, so that the effects of the synchronization overhead such as the cost of additional bit storage, execution latency or extra network traffic can be studied in detail.

4. FGS Snoopy Coherence Protocol

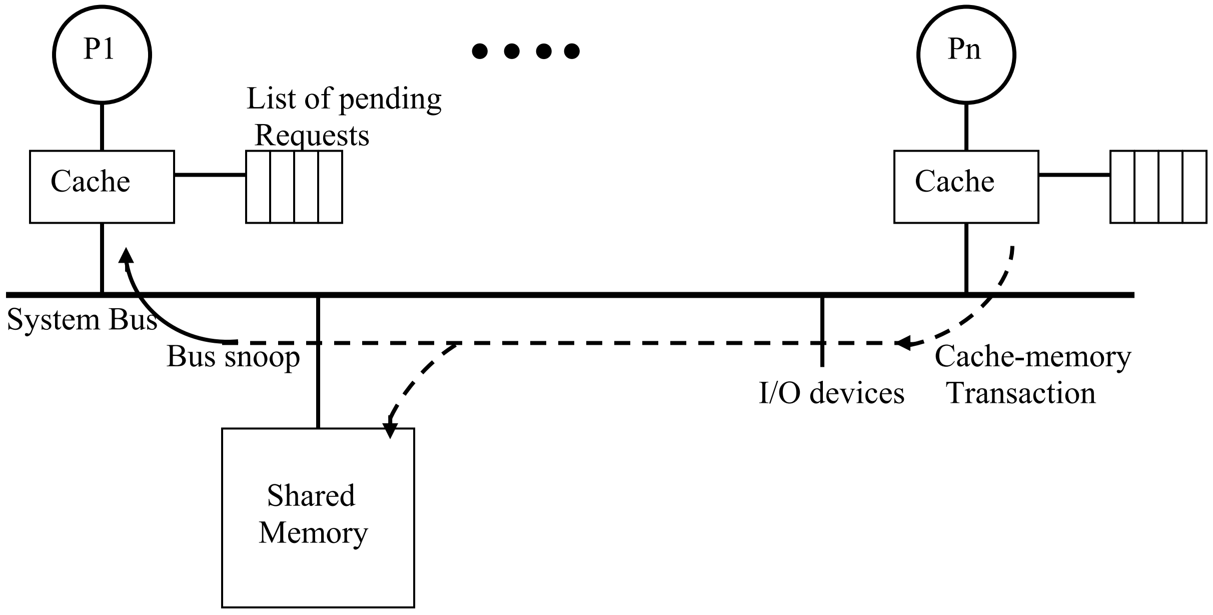


Figure 7: Snoopy cache-coherent multiprocessor with Shared-Memory

Bus-based system architecture, figure 7, illustrates the bus connection of processing nodes with their private caches placed on a shared bus. Each processing node's cache controller continuously *snoops* on the bus watching for relevant transaction and updates its state suitably to keep its local cache coherent [12]. The dashed-line and arrows shows the transaction being placed on the bus and accepted by main memory as in uniprocessor system. The continuous line shows the *snoop.* The key properties of the bus that support coherence are the following:

- All transactions that appear on the bus are visible to all cache controllers.
- They are visible to all controller in the same order (the order in which they appear on the bus)

A coherence protocol must guarantee that all the "necessary" transactions appear on the bus, in response to memory operations, the controllers should take the appropriate actions when they see a relevant transaction. The protocol described here is based on the *MESI protocol*, also knows as *Illinois protocol.* It is a four-state write-back invalidation protocol with the following state semantics [12]:

- *Modified* – cache has the valid copy of the block and location in main memory is invalid.
- *Exclusive clean* – cache has a copy of the block and main memory is up-to-date. A signal is available to the controller in order to determine on a *BusRd* if any other cache currently holds the data.
- *Shared* – the block is present in an unmodified state in the cache and zero or more caches may also have a shared copy, main memory is up-to-date
- *Invalid* – no valid data is present in the block.

The state transition diagram of *MESI* protocol without fine-grain synchronization support is shown in Figure 8. The notation A/B means that 'A' indicates an observed event whereas 'B' is an event generated as a consequence of A. Dashed lines show state transitions due to observed bus transactions, while continuous lines indicate state transitions due to local processor actions.

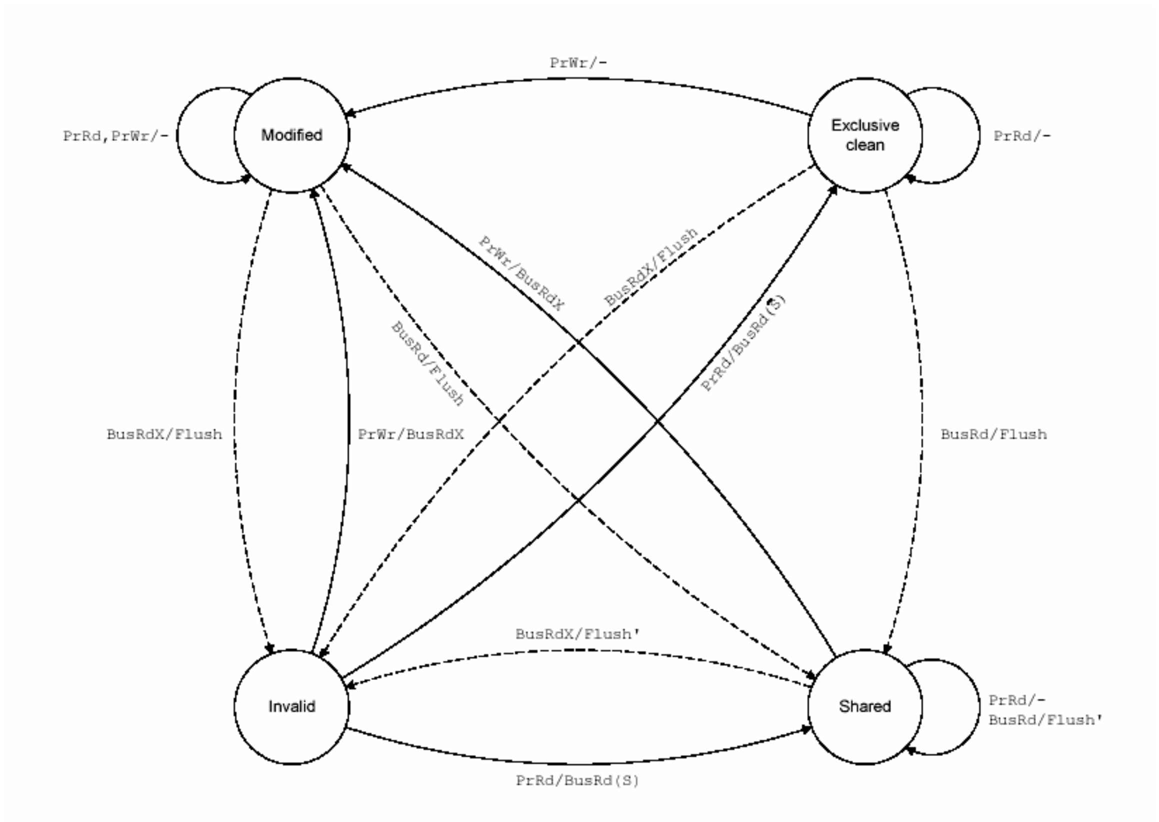


Figure 8: MESI cache coherence protocol

Finally, the notation Flush means that data is supplied only by the corresponding cache. Also this diagram does not consider the transient states used for bus acquisition.

4.1 Protocol Description

The state transitions needed to integrate fine-grain synchronization in *MESI* can be done by splitting the ordinary *MESI* sates into two groups: *empty state* transitions and *full state* transitions. In the protocol description, we consider only waiting non-altering reads and waiting altering writes. Altering reads can be achieved by issuing non-altering reads in combination with an operation that clears the FE-bit without retrieving data. This operation is called as unconditional altering clear (PrUACl) and it operates on a *FE-bit* without accessing or altering the data corresponding to that state bit. In order to reuse synchronized memory locations, clearing of *FE- bits* is necessary (this is described in detail in [20]). This operation can be initialized as soon as there is no pending read for that location (Pr-bit is clear) and FE-bit need to be reset to reuse that memory location.

The most complex synchronizing operations in cache are the waiting read/write operations because they require additional hardware in order to manage deferred list and resume pending synchronization requests. The rest of the synchronizing operations are simpler version of waiting read/write operations with the only difference being in the behavior of operations, when a synchronization miss is detected. Instead of adding the rest of these synchronizing operations in pending list, either an exception is raised or the operation is discarded.

Two additional bus transactions have been introduced in order to integrate fine-grain synchronization with cache coherence in the MESI protocol [27], which ensures the coherence of FE-bits and Pending-bits. Table 4 describes in more details.

Table 4: Additional bus transactions in the MESI protocol

Bus Transaction	**Description**
BusSWr	A node has performed an *altering waiting write* and reset the *Pr-bit*. The effect of this operation in observing nodes is to set the *FE-bit* and reset the *Pr-bit* of the referring memory location to resume the relevant pending-read requests. If more than one pending-write is there for that memory location then *Pw-bit* need to set again after completing the altering waiting write.
BusSCl	A node has performed an *altering read* or an *unconditional clear* operation. The effect of this operation in observing nodes is to clear the *FE-bit* and reset the *Pw-bit* of the referring memory location, thus making it reusable.

The new bus signal ‘C’ is introduced to determine the condition of synchronized operation miss, named *shared-word* signal and indicates if there is any other node sharing to the specified word. This signal can be implemented as a wired-OR controller line, which is asserted by each cache that contains the copy of the relevant word with the *FE-bit* set.

It is necessary to specify the particular data word on which synchronization operation is performed because cache line may contain several synchronized data words. A negated signal (C') causes a requesting read to be appended to the list of pending reads in *MSHR,* sets the *Pr-bit* (if not set*)* and resets the *Pw-bit* to resume pending-writes (if any), otherwise perform the new incoming requesting writes. If the synchronization signal 'C' is asserted, then it resets the *Pr-bit* to resume the pending-reads (if any), otherwise new synchronized read is processed and requesting write is appended to the list of pending writes in *MSHR* and the *Pw-bit* is set.

Along with *shared-word* signal which is already introduced, three more wired-OR signals are required for the protocol to operate correctly [12]. The first signal (named S) is asserted if any processor different than the requesting processor has a copy of the cache line. The second signal is asserted if any cache has the block in a dirty state. This signal modifies the meaning of the 'S' signal in the sense that an existing copy of a cache line has been modified and then all the copies in other nodes are invalid. A third signal is necessary in order to predict whether all the caches have completed their snoop, which means, it is reliable to read the value of the first two signals.

4.2. Correspondence between processor instructions and bus transactions

When a processing node issues any memory operation, the local cache first interprets the request and then performs accordingly, if required it also issues the bus transaction. The correspondence between the different processor instructions and the memory requests seen on the bus is shown in following Table 5.

Table 5: Correspondence between processor instructions and memory requests

Request from processor	**Bus transaction**
PrUNRd	BusRd (Ordinary read)
PrUNWr	BusWr (Ordinary write)
PrUARd	BusRd + BusSCl
PrUAWr	BusAWr (Not specified in protocol definition)
PrWNRd	BusRd (C) (Bus transaction with shared-word signal)
PrWNWr	BusWr (C)
PrWARd	BusRd (C) + BusSCl
PrWAWr	BusSWr (C)

From Table 5, it can be inferred that unconditional read/write requests from the processor generates the ordinary read/write transaction on the bus. Unconditional altering read PrUARd, requires BusRd transaction followed by BusSCl transaction, therefore this request retrieves the data from the corresponding memory location and as well clears the FE-bit. Clearing the FE-bit is performed by the BusSCl transaction, which does not access nor modifies the data. Finally unconditional write request PrUAWr, generates the bus transaction, namely BusAWr, which unconditionally sets the FE-bit after writing the corresponding data to the specified memory location.

Table 5 shows that the behavior of all the conditional memory operation depends on the shared-word bus signal. A conditional non-altering read operation generates an ordinary read bus transaction after checking the status of shared-word signal, if it is asserted. A conditional altering read operation generates ordinary read transaction in addition to the BusSCl transaction. Finally, a conditional altering write causes a BusSWr transaction to be initiated on the bus. This transaction sets the FE-bit and resets the Pr-bit after writing the corresponding data to the referred memory location to resume the pending-read operations, if any exists.

4.3. Resuming of pending requests

It is very crucial to specify how the resuming of pending requests is done. In the *snoop-based* systems, coherence of *FE-bit* and *Pending-bits* is ensured by the proper bus transactions. It means that all caches those have *pending* read/write requests for a given memory location will get to know when the synchronization condition is met by snooping into the bus and monitoring for the BusSWr or BusSCl transactions to occur.

When any bus transaction occurs, a comparator in a cache checks if there is any entry in *MSHR* matching with the received bus transaction. If incoming transaction matched with any *MSHR* entry and bus transaction is *BusSWr*, then the observing node will perform altering-write operation, it will set the *FE-bit* and reset the *Pr-bit* to resume the pending read for the referred location. On the other hand, if bus transaction is *BusSCl* then the observing node will perform altering-read or unconditional clear operation, it will reset the *FE-bit* and *Pw-bit* to resume the pending write for the referred location.

It is also possible to have pending requests for the memory location that is not cached or is in invalid state. The location will be cached in the cache as soon as the synchronization miss is resolved to make it available for the processing at the desired node.

Considering the example of three node bus system shown in Figure 9, and assume that every node has pending requests for location 'X' in their respective *MSHR*. Suppose nodes A and B have invalid copies in their caches along with *Pr-bit* is set (means read request is pending for the location ‘X’), whereas node C has the exclusive ownership of the referred location ‘X’, whose *FE-state bit* and *Pw-bit* is unset. After node C successfully performs a conditional altering write to location ‘X’ and unset the *Pr-bit* to resume pending read, if any available at this node for the location ‘X’ as well this event is notified on the bus by a BusSWr transaction.

This transaction informs nodes A and B that they can reset their *Pr-bit* corresponding to the location ‘X’ to resume the pending read requests, which happens to be a conditional altering read. As a consequence, only one of these nodes will be able to successfully issue the operation at this point. This is imposed by bus order. For instance, if node B gets the bus ownership before node A, the pending request from the node B will be resumed first and the operation at node A will stay pending in the MSHR.

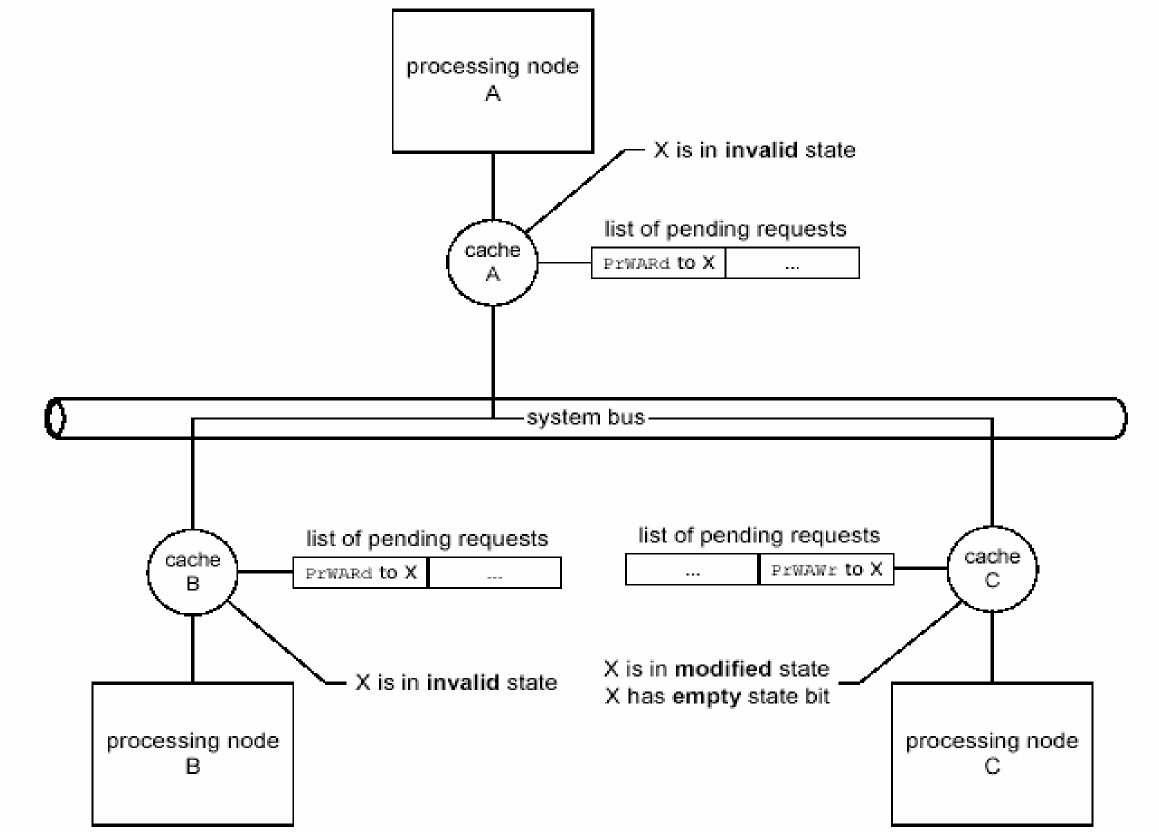


Figure 9: Resuming of pending requests.

While handling multiple *pending write* requests for a single memory location, the cache controller analyzes the tag along with *FE-bit* and *pending read/write bits* for the new synchronized write operation and if write miss occurs and *Pw-bit* is already set (means already there is a pending write for that location). In this case, the later synchronized write miss will be added to the deferred list of *MSHR* and it will be linked with the former synchronized write miss to the same location. As soon as all the synchronized read misses will be resolved for this memory location, the cache controller will reset the *FE-bit* and Pw-*bit* to resume the pending write miss. The very first pending synchronized write miss will be activated and will perform the write operation It will set the *FE-bit*, reset the *Pr-bit* to resume the pending reads and at the same time sets the *Pw-bit* again to take care the write misses for the same location, those are still pending to resolve.

4.4. Transition rules of Synchronized Snoopy-based protocol

Transition rules from each coherence state are presented in the following sections for the four state *MESI* protocol. These transition rules are similar to those described in [27] but each rule is modified in order to capture the handling of synchronized pending read/write operations and their deferred list. A description made here is in the form of C-styled pseudo-code for the each state. It explains how transition happens from one state to other. It is noted that the ordering of all kind of misses (cache misses and synchronized misses) from different processors is maintained by the bus order.

4.4.1. Transition from the Invalid State

```
SWITCH (IncomingRequest) {
   //Processor Requests
   CASE PrUNRd : Send (BusRd);
                 IF (S) {
                    FlushFromOtherCache(); NextState = Shared;
                 }  ELSE {
                    ReadMemory( );NextState = Exclusive;
                 }  Break;
   CASE PrUNWr: Send (BusRdX); NextState = Modified; Break;
   CASE PrWNRd: Send (BusRd);
                IF (S && C) {
                    FlushFromOtherCache(); NextState = Shared;
                 }  ELSE IF(!S && C) {
                    ReadMemory(); NextState = Exclusive;
                 }  ELSE {
                    AddToDeferredList();  //Wait to resolve.
                    SetPrBit ( ); NextState = Invalid;
                 }  Break;
   CASE PrWAWr : Send ( BusWr );
                 IF (S && !C) {
                    WriteToBus(); NextState = Shared;//To resolve
                 }  ELSE IF (!S && !C) {
                    WriteToCache(); NextState = Modified;
                 }  Else {
                    AddToDeferredList(); // Wait to Resolve.
                    SetPwBit ( ); NextState = Invalid;
                 }  Break;
   CASE PrUACl : IF (C){
                    Send ( BusSCl ); NextState = Invalid;
                 }  Break;
```

4.4.2 Transition from the Modified State

```
SWITCH (IncomingRequest)
  {
  //Processor Requests
  CASE PrUNRd: ReadCache(); NextState = Modified; Break;
  CASE PrUNWr: WriteToCache(); NextState = Modified; Break;
  CASE PrWNRd: IF(Full) {
                  ReadCache(); NextState = Modified;
               }  ELSE {
                  AddToDeferredList();         // Wait to Resolve
                  SetPrBit(); NextState = Modified;
               }  Break;
  CASE PrWAWr: Send(BusSWr);
               IF(Empty) {
                  WriteToCache();NextState = Modified;
                  ResetPrBit();            //Resume pending reads
               }  ELSE {
                  AddToDeferredList();         // Wait to resolve
                  SetPwBit();NextState = Modified;
               }  Break;
  CASE PrUACl: IF (Full) {
                  ReSetFE();NextState = Modified;
                  ReSetPwBit();            //Resume pending write
               }  Break;

  ---- Bus Signals
  CASE BusRd:  flush(); NextState = Shared; Break;
  CASE BusRdX: flush(); NextState = Invalid; Break;
  CASE BusSWr: IF(Empty) {
                  WriteToCache(); NextState = Shared;
                  UnSetPrBit();            //Resume pending reads
               }  Break;
  CASE BusSCl: IF(Full) {
                  ReSetFE(); NextState = Shared;
                  ReSetPwBit();            //Resume pending write
               }  Break;
  }
```

4.4.3 Transition from the Exclusive State

```
SWITCH (IncomingRequest)
  {
  //Processor Requests
  CASE PrUNRd: ReadCache(); NextState = Exclusive; Break;
  CASE PrUNWr: WriteToCache(); NextState = Modified; Break;
  CASE PrWNRd: IF(Full) {
                  ReadCache();NextState = Exclusive;
               }  ELSE {
                  AddToDeferredList();          // wait to resolve
                  SetPrBit(); NextState = Exclusive;
               }  Break;
  CASE PrWAWr: Send(BusSWr);
               IF(Empty) {
                  WriteToCache();
                  ReSetPrBit();             //resume pending reads
                  NextState = Shared;       // need to evaluate
```

```
             }  ELSE {
                AddToDeferredList();      //wait to resolve
                SetPwBit();NextState = Exclusive;
             }  Break;
CASE PrUACl: IF(Full) {
                ReSetFE(); NextState = Modified;
                ReSetPwBit();   //resume pending write
             }  Break;

//// Bus Signals
CASE BusRd:  flush(); NextState = Shared; Break;
CASE BusRdX: flush(); NextState = Invalid; Break;
CASE BusSWr: IF(Empty) {
                WriteToCache(); NextState = Shared;
                ReSetPrBit();             //resume pending reads
             }  Break;
CASE BusSCl: IF(Full) {
                ReSetFE();NextState = Shared;
                ReSetPwBit();             //resume pending write
             }  Break;
}
```

4.4.4 Transition from the Shared State

```
SWITCH (IncomingRequest)
  {
  //// Processor Requests
  CASE PrUNRd: ReadCache();NextState = Shared; Break;
  CASE PrUNWr: Send(BusRdX); WriteToCache();NextState = Modified;
               Break;
  CASE PrWNRd: IF(Full) {
                  ReadCache();NextState = Shared;
               }  ELSE {
                  AddToDeferredList();          //wait to resolve
                  SetPrBit(); NextState = Shared;
               }  Break;
  CASE PrWAWr: Send(BusSWr);
               IF(Empty){
                  WriteToCache();
                  ReSetPrBit();            //resume pending reads
                  NextState = Shared;         // need to evaluate
                }  ELSE {
                   AddToDeferredList();        //wait to resolve
                   SetPwBit(); NextState = Shared;
                }  Break;
  CASE PrUACl:  IF(Full){
                   ReSetFE();
                   ReSetPwBit();           //resume pending write
                   Send(BusSCl); NextState = Shared;
                }  Break;

  ///// Bus Signals
  CASE BusRd:   Flush(); NextState = Shared; Break;
  CASE BusRdX:  Flush(); NextState = Invalid; Break;
  CASE BusSWr:  IF(Empty){
                   WriteToCache();NextState = Shared;
```

```
                    ReSetPrBit();          //resume pending reads
                 }  Break;
    CASE BusSCl:  IF(Full){
                    ReSetFE(); NextState = Shared;
                    ReSetPwBit();          //resume pending write
                 }  Break;
    }
```

4.5. Merging of pending requests

Each processing node maintains its local deferred list. This list contains both cache misses and synchronization misses. It can happen that both types of misses are for the same location. So in this case, not only cache line is present but also synchronization state is not met at the location where the copy of word is present. After a relevant change in the synchronization state of the memory location, any operation that matches with the present synchronized state is resumed at the appropriate processing node.

Table 6 shows how the management of the deferred list is done at a local node. More precisely, this table specifies the action taken by cache controller when a new request is received with respect to a pending request already present in the list of deferred operations. This merging of requests is basically a optimization in coherence protocol. It is noted that a pending write is always conflicting with any incoming requests, so it can never be merged and required separate entry in list of pending requests. Since write operations are always conflicting, all write requests have not shown in the Table 6.

Table 6: Merging of requests with incoming requests.

Incoming requests	**Can be merged with**
PrUNRd	Any previous pending read request referred to the same location.
PrUARd	
PrWNRd	Only non-altering pending read request referred to the same location.
PrWARd	
PrNNRd	
PrNARd	
PrTNRd	
PrTARd	

From the table 6, it can be observed that incoming unconditional read operations can be merged with any pending reads requests whereas incoming conditional reads can be merged with only non-altering pending read requests.

4.6. Discussion

A bus based snoopy coherence protocol integrated with fine-grain synchronization support has been introduced. This implementation considers only waiting non-altering reads, waiting altering writes and unconditional clearing *FE-bits* memory operations. A systematic protocol description is made here in the form of state transitions and their corresponding C-styled pseudo-code.

In the snoopy-protocol the coherence of *FE-state bits* and *Pending- bits* is maintained by bus transactions defined for this purpose, namely *BusSWr* and *BusSCl*. An additional bus signal 'C' called *shared-word* along with three more bus signals are also introduced in order to implement the conditional behavior of synchronizing operations.

A drawback of integrating fine-grain synchronization with cache coherence at the cache level is the complexity of managing and resuming of pending synchronization requests. This complexity can be over come by the use of aggressive and efficient hardware support in the system in terms of *FE-bits* and *Pending-bits,* which can exploit the efficiency of fine-grain synchronization to achieve high degree of parallelism and improve its performance.

5. FGS directory-based coherency protocol

Scalable cache coherence is typically based on the concept of a directory. Since the state of a block in a cache can no longer be determined implicitly by placing a request on a shared bus and having it snooped by the cache controllers, the idea is to maintain the state explicitly in a place – called a *directory*. It can be imagine that each cache-line-sized block of main memory has associated with it a record of the caches that currently contain a copy and the state of the block in those caches. This record is called the *directory entry* for that block.

Figure 10, shows a system of a scalable multiprocessor with directories. 'CA' works as communication assist between cache, main memory and interconnection network. All the endpoint processing at the destination of the transaction (invalidating blocks, retrieving and replying with data) is typically done by the communication assist rather than the main processor.

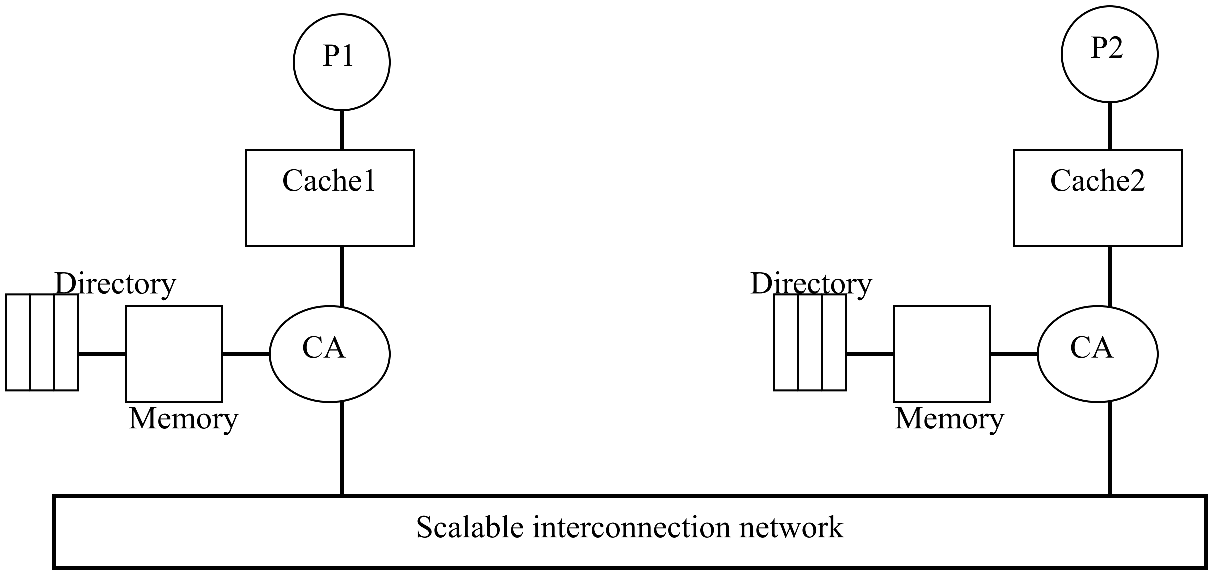


Figure 10: A scalable multiprocessor system with directories

Since directory schemes rely on *point-to-point* network transactions, they can be used with any interconnection network. In a scalable multiprocessor based system, shown below in Figure 10, each shared memory block has a directory entry that lists the nodes that have a cached copy of the data [12]. *FE-bit* and *Pending-bits* are stored as an extra field on the coherence directory entry to implement efficiently the synchronization conditions. *Point-to-point* messages are used to keep the directory *up-to-date* and to request permissions for a *load* and *store* to a particular location.

5.1. Alewife directory based coherence protocol

The following directory states are defined in *Alewife* machine for the coherence protocol:

- *Read-only:* One or more caches have a read-only copy of the block
- *Read-write:* Only one cache has a read-write copy of the block
- *Read Transaction:* Cache is holding a read request (update is progress)
- *Write transaction:* Cache is holding a write request (invalidation is in progress)

5.2. Directory modification to support FGS

All the caches and directory blocks contain an array of the *FE-bits* – one for each word. Each cache and directory block is also tagged with pending bits (Pr-bit and Pw-bit). If a *Pr-bit* is set, it means there is pending synchronized read for the corresponding word. This information is required so that synchronized write can immediately resume the pending read after it has completed writing at the corresponding memory location.

On the contrary, if *Pw-bit* is set, it means there is a pending synchronized write and this information is required so that synchronized read can immediately resume the pending write after finishing the reading to the corresponding memory location. If more than one synchronized write are pending for the same location, then will have separate entries for each pending write and *Pw-bit* will remain set until all writes resolved. This will be taken care by the directory controller, which will keep observing the pending list of synchronized write misses.

The home node (directory) also contains a *Synchronization Miss Buffer (SMB),* which holds information regarding which node has pending synchronized read/write for a given word. Figure 11 shows the modified directory and SMB organization for a 4 processor system with 32 byte memory blocks (4 words, a example).

It has been already mentioned that SMB contains entries of nodes which suffered from synchronization read/write miss and entry is indexed by the word address. But how many entries should this SMB contain? For *hit-under-miss* architecture with 'n' nodes, there can be utmost 'n–1' pending operations [26]. For very large configuration or for *miss-under-miss* architecture, the requisite number of entries can be very large.

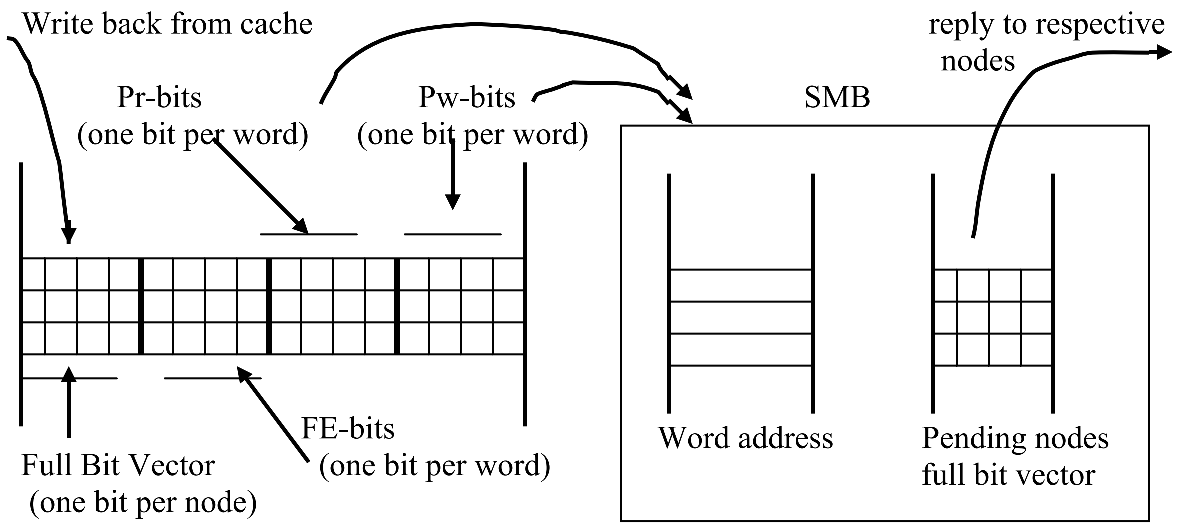


Figure 11: Modified directory and SMB

In such cases, an overflow mechanism can be employed – if entries are running out of SMB entries, an overflow bit can be set and the directory controller would treat this situation differently. Either directory controller would assume that all nodes are pending [26] or it will replace any cache randomly with upcoming cache [27]. In former method, controller will send reply to all of them accordingly depending on the kind of request. During this time processor will be stalled and directory controller will not accept anymore request until it would not get any SMB entry free.

All synchronization misses are kept in *MSHR* of the remote node to be subsequently resolved by the explicit messages from the home node (directory). The cache controller has to match not only tag but also the state bits depending upon the instruction and take decision based on the state of the cached line as well as the associated *FE-bit* and *Pending read/write bits.* The directory controller is very complicated because of the *FSM*, which implements the s*ynchronization coherence protocol* in the directory. It has to send data asynchronously to resolve synchronization misses on write backs by looking up *SMB* and so on.

5.3. Correspondence between processor instructions and network transaction

The network transactions used in the proposed protocol are described in Table 7. It shows the requests sent from cache to memory and requests sent from memory to cache. Six new messages are introduced in order to implement fine-grain synchronization at the cache level. These messages are SRREQ, SWREQ, SCREQ from cache to memory and SRDENY, SWDENY and ACKSC from memory to cache.

Table 7: Network transactions in the directory-based protocol

Type of Message	**Symbol**	**Semantics**
Cache to Memory	RREQ	Request to read a word that is not in the cache
	WREQ	Request to write a word
	SRREQ	Waiting and non-altering read request
	SWREQ	Waiting and altering write request
	SCREQ	Request to clear FE-bit
	UPDATE	Returns modified data to memory
	ACKC	Acknowledges that a word has been invalidated
Memory to Cache	RDATA	Contains a copy of data in memory (response to RREQ)
	WDATA	Contains a copy of data in memory (response to WREQ)
Memory to Cache	SRDENY	Sent if a SRREQ misses and request is appended to pending list at directory
	SWDENY	Sent if a SWREQ misses and request is appended to pending list at directory
	INV	Invalidates the cache words
	ACKSC	Acknowledges that the FE-bit has been unset in all the copies of the block
	BUSY	Response to any RREQ or WREQ while invalidation are in progress

SRDENY and *SWDENY* transaction issues from the memory when synchronization miss occurs, directory controller creates and entry at *SMB* corresponding to the miss and controller resolves it when synchronization condition meets for the specified memory location. Following Table 8 shows that how cache interprets the request from processor and translates it into one or more network transactions.

Table 8: Correspondence between processor instructions and network transaction

Instruction from Processor	**Initiated network transaction**
PrUNRd	RREQ
PrUNWr	WREQ
PrUARd	RREQ + SCREQ
PrUAWr	-
PrSNRd	SRREQ
PrSNWr	CWREQ
PrSARd	SRREQ + SCREQ
PrSAWr	SWREQ

5.4. Directory transition rules

The following description of transition states is based on the *Alewife coherence protocol* [9]. The rules defined here are similar to those described in [26], [27]. But each rule is modified and extended in order to manage pending bits for synchronized read/write and SMB entries. Transition rules in each coherence state are presented with their C-styled pseudo-code for the *directory-based* protocols. The protocol is completely defined in following five states. In this description, directory controller will replace any cache randomly with the new upcoming cache in case of overflow at SMB entries.

5.4.1. Transition from the Absent State

```
SWITCH (IncomingRequest) {
 CASE RREQ(i): addNodeToDirectory(i); //"i" is the sending node id
               Send (RDATA, i);      //send requested data to node
               NextState = ReadOnly; BREAK;
CASE WREQ(i): IF (ackCounter == 0) {
                  addNodeToDirectory(i); Send (WDATA,i);
                  NextState = ReadWrite;
              }  ELSE {
                  addNodeToDirectory(i);
                  NextState = WriteTransaction;
              }  BREAK;
CASE SRREQ(i): IF(FULL) {
                  addNodeToDirectory (i); Send (RDATA, i);
                  NextState = ReadOnly;
               }  ELSE {
                  Send (RDENY,i); AddToDeferredList();
                  SetPrBit ( ); NextState =Absent;
               }  BREAK;
CASE SWREQ(i): IF(EMPTY && deferredListEmpty()){
                  addNodeToDirectory (i); Send (WDATA, i);
                  NextState = ReadOnly;
               }  ELSE IF (EMPTY &&! deferredListEmpty()){
                  addNodeToDirectory(i); Send (WDATA,i);
                  ResetPrBit(); NextState = ReadOnly;
               }  ELSE {
                  Send (WDENY,i); addToDeferredList();
                  SetPwBit(); NextState = Absent;
               }  BREAK;
 CASE SCREQ(i): ResetFE(); Send (ACKSC,i); ResetPwBit();
                NextState = Absent; BREAK;
 CASE ACKC(i):  ackCounter--; NextState = Absent; BREAK;
}
```

5.4.2. Transition from the Read-only state

```
SWITCH (IncomingRequest) {
  CASE RREQ(i): IF(hasPointerInDirectory(i)) {
                   Send (RDATA, i); // "i" is the sending node id.
                }  ELSE IF(!directoryFull( )) {
                   addNodeToDirectory(); Send(RDATA,i);
                }  ELSE {
                     ++ackCounter; j = evictRandomDirectoryEntry();
                     Send(INV,j); addNodeToDirectory();
                     Send (RDATA, i);
                  }  NextState = ReadOnly; BREAK;
CASE WREQ(i): IF(hasPointerInDirectory(i) && (numberOfEntries()>1)){
                     ackCounter += numberOfEntries() - 1;
                     FOR(j = 0; j < numberOfEntries(); j++) {
                     IF (i != j) Send (INV, j);
                     } clearDirectory ( ); addNodeToDirectory (i);
                     NextState = WriteTransaction;
                  }  ELSE IF(hasPointerInDirectory(i) &&
                            (numberOfEntries () == 1) &&
                            (ackCounter != 0)) {
                     NextState = WriteTransaction;
                  }  ELSE IF(hasPointerInDirectory(i) &&
                            (ackCounter == 0)) {
                     Send(WDATA, i); NextState = ReadWrite;
                  }  ELSE {   // if the line is not in the directory
                     ackCounter += n;
                     FOR(j = 0; j < numberOfEntries(); j++) {
                     Send (INV, j);
                  }  clearDirectory ( ); addNodeToDirectory (i);
                  }  BREAK;
CASE SRREQ(i: IF(FULL && hasPointerInDirectory(i)) {
                     Send (RDATA, i);
                  }  ELSE IF(full && ! directoryFull()) {
                     addNodeToDirectory(); Send (RDATA,i);
                  }  ELSE IF(FULL && directoryFull()) {
                     ++ackCounter;j = evictRandomDirectoryEntry();
                                           // j is the evicted line.
                     Send(INV,j); addNodeToDirectory();
                     Send(RDATA,i);
                  }  ELSE(EMPTY) {
                     Send(RDENY,i); addToDeferredList();
                     SetPrBit();            //synchronized read miss
                  }  NextState = ReadOnly; BREAK;
CASE SWREQ(i): IF(EMPTY && deferredListEmpty()) {
                     addNodeToDirectory (i); Send (WDATA, i);
                  }  ELSE IF(EMPTY && ! deferredListEmpty()) {
                     addNodeToDirectory(i); Send(WDATA,i);
                     ResetPrBit();             //resume pending read
                  }  ELSE {
                     Send(WDENY,i); addToDeferredList();
                     SetPwBit();           //synchronized write miss
                  }  NextState = ReadOnly; BREAK;
CASE SCREQ(i):  IF(numberOfEntries() > 1) {
                     ackCounter += numberOfEntries() - 1;
                     FOR(j = 0; j < numberOfEntries();j++) {
                     IF(i != j) Send(SCREQ,j);
                     } clearDirectory();
                     addNodeToDirectory();
                     NextState = WriteTransaction;
                  }  ELSE IF(hasPointerInDirectory(i)) {
                     ResetFE();                //clearing the FE-bit
                     ResetPwBit();             //clearing the Pw-bit
                     Send (ACKSC, i);NextState = ReadOnly;
                  }  BREAK;
CASE ACKC(i):   ackCounter--; NextState = ReadOnly; BREAK;
}
```

5.4.3. Transition from the Read-write state

```
SWITCH(IncomingRequest) {
  CASE RREQ(j): IF(!hasPointerInDirectory(j)){ //there is only one
                  ++ackCounter;              // node in directory
                  Send (INV, i);         //(the owner ,namely ’i’)
                  ClearDirectory(); addNodeToDirectory(j);
                  NextState = ReadTransactoin;
                } BREAK;
  CASE WREQ(j): IF(!hasPointerInDirectory(j)) {
                  ++ackCounter; Send (INV,i); clearDirectory();
                  addNodeToDirectory (j);
                  NextState = WriteTransaction;
                } BREAK;
 CASE SRREQ(j): IF(!hasPointerInDirectory(j) && FULL) {
                  ++ackCounter; Send(INV,i); clearDirectory();
                  addNodeToDirectory(j);
                  NextState = ReadTransaction;
                } ELSE IF(EMPTY) {
                  Send (RDENY,j); addToDeferredList();
                  SetPrBit();             //synchronized read mis
                  NextState = ReadWrite;
                } BREAK;
 CASE SWREQ(j): IF(!hasPointerInDirectory(j) && EMPTY) {
                  ++ackCounter; Send(INV,i); clearDirectory();
                  addNodeToDirectory(j);
                  ResetPwBit();             //resume pending write
                  NextState = WriteTransaction;
                } ELSE IF(FULL) {
                  Send(WDENY,j); addToDeferredList();
                  SetPwBit();             //synchronize write miss
                  NextState = ReadWrite;
                } BREAK;
 CASE SCREQ(j): Send(SCFWD,i); NextState = ReadWrite; BREAK;
 CASE ACKC(j):  ackCounter--; NextState = ReadOnly; BREAK;
 CASE UPDATE(i,Dpack): addToDeferredList(Dpack);
                       ResetPrBit();        //resume pending read
                       NextState = ReadOnly; BREAK;
 }
```

5.4.4. Read transaction state

```
SWITCH(IncomingRequest) {
  CASE RREQ(i):  Send(BUSY,i); NextState = ReadTransaction; BREAK;
  CASE WREQ(i):  Send(BUSY,i); NextState = ReadTransaction; BREAK;
  CASE SRREQ(i): Send(BUSY,i); NextState = ReadTransaction; BREAK;
  CASE SWREQ(i): Send(BUSY,i); NextState = ReadTransaction; BREAK;
  CASE SCREQ(i): Send(BUSY,1); NextState = ReadTransaction; BREAK;
  CASE ACKC(i):  ackCounter--; NextState = ReadOnly; BREAK;
  CASE UPDATE(i): --ackCounter; Send(RDATA,i);
                  NextState = ReadOnly; BREAK;
 }
```

5.4.5. Write transaction

```
SWITCH (IncomingRequest) {
 CASE RREQ(i):  Send(BUSY,i); NextState = WriteTransaction; BREAK;
 CASE WREQ(i):  Send(BUSY,i); NextState = WriteTransaction; BREAK;
 CASE SRREQ(i): Send(BUSY,i); NextState = WriteTransaction; BREAK;
 CASE SWREQ(i): Send(BUSY,i); NextState = WriteTransaction; BREAK;
 CASE SCREQ(i): Send(BUSY,i); NextState = WriteTransaction; BREAK;
 CASE ACKC(i):  IF(ackCounter == 1) {
                   ackCounter = 0; Send(WDATA,cacheInDirectory());
                   NextState = ReadWrite;
                }  ELSE {
                   --ackCounter; NextState = WriteTransaction;
                }  BREAK;
 CASE ACKSC(i): IF(ackCounter == 1) {
                   ackCounter = 0; Send(ACKSC,cacheInDirectory();
                   NextState = ReadWrite;
                }  ELSE {
                   --ackCounter; NextState = WriteTransaction;
                }  BREAK;
 CASE UPDATE(i): IF(ackCounter == 1) {
                   ackCounter = 0; Send(WDATA,cacheInDirectory());
                   NextState = ReadWrite;
                }  ELSE {
                    --ackCounter; NextState = WriteTransaction;
                 }  BREAK;
  }
```

5.5. Discussion

A directory based protocol has been defined to integrate fine-grain synchronization with cache coherence protocol, and aggressive and efficient hardware approach has been discussed to support this protocol. The protocol has been systematically specified in the form or *C-styled* pseudo code for each state. Only waiting non-altering read, waiting altering-write and unconditional clearing *FE-bit* operations have been discussed whereas waiting altering read can be produced in the combination with non-altering read and unconditional clearing *FE-bit* operation.

Six new network messages are introduced in the protocol to integrate fine-grain synchronization with cache coherence. In the proposed protocol definition, synchronized write is not suspended upon synchronization miss, whereas it is treated as ordinary cache miss and appended to deferred list. This miss is resolved accordingly when the synchronization condition is met. Multiple synchronized write misses for the same location are stored as a linked list at the home directory and resolved one by one in a sequence. We suggest that a deferred list management scheme in which list of pending requests can be either at home directory or distributed between home and caches. Trade off between *centralized* design and *distributed* approach can be done to get optimized protocol definition.

6. Evaluation Framework

Refer [27], the evaluation of the proposed ideas, in particular directory-based protocol is done via simulation of a multiprocessor system supporting the fine-grain synchronization. In previous work [27], simulation was done using RSIM, an event drive simulator [17], [18] and some simple applications were used in simulation experimentation.

In this project, we use *SimpleScalar* Simulator to evaluate the performance of proposed architecture along with the proposed protocol definition in *bus-based* and *directory-based* model. Since the source code of simulator does not support the proposed protocol definition and the features of the suggested architectural model of the system so it is needed to modify accordingly to support the suggested features.

It is also necessary to simulate the standard application like *MICCG3D* from *SPLASH* 2 using SimpleScalar. After the modifying in the source code of simulator, evaluate the performance and compare the obtained results with the standard results available from *MIT Alewife* Machine for the corresponding application. That will demonstrate the performance of proposed architectural model which is very much based on hardware approach against the software approach adopted in *Alewife Machine*.

6.1. SimpleScalar Simulator

The SimpleScalar tool set [23] is a system software infrastructure used to build modeling applications for program performance analysis, detailed micro-architectural modeling, and hardware-software co-verification. Using the SimpleScalar tools, users can build modeling applications that simulate real programs running on a range of modern processors and systems [14].

The tool set includes sample simulators ranging from a fast functional simulator to a detailed, dynamically scheduled processor model that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction. In addition to simulators, the SimpleScalar tool set includes performance visualization tools, statistical analysis resources, and debug and verification infrastructure.

SimpleScalar simulators can emulate the *Alpha, PISA, ARM*, and *x86* instruction sets. The tool set includes a machine definition infrastructure that permits most architectural details to be separated from simulator implementations. The current release of SimpleScalar can run programs from any of the above listed instruction sets. Complex instruction set emulation (e.g., x86) can be implemented with or without microcode, making the SimpleScalar tools particularly useful for modeling *CISC* instruction sets.

The *PISA* instruction set (Portable Instruction Set Architecture) is a simple *MIPS*-like instruction set maintained primarily for instructional use. A GNU *GCC*-based cross-compiler and *pre-built* libraries are also available for this target. The *PISA* target is particularly useful for computer engineering instruction as the tools can be built on a wide range of host platforms, including *Linux/x86, Win2000, SPARC Solaris*, and others.

SimpleScalar builds on most 32-bit and 64-bit flavors of *UNIX* and *Windows NT*-based operating systems. The internal software architecture of the tool set includes a host interface module, permitting fast porting to other host platforms. The host interface module permits *cross-endian* emulation, thus it is possible to use emulate a target on a host platform with a different endian, *e.g.*, running Alpha *ISA emulation* on a *SPARC* Solaris host platform.

6.2. SimpleScalar tool set overview

Figure 12 shows the graphical overview of *SimpleScala*r tool set [14]. Benchmarks written in FORTRAN are converted to C using Bell Labs' *f2c* converter. Benchmarks, one written in 'C' and other converted from FORTRAN into 'C' are complied using the *SimpleScalar* version of GCC, which generates SimpleScalar assembly. The SimpleScalar *assembler* and *loader*, along with the necessary ported libraries, produce SimpleScalar executables that can then be fed directly into one of the provided simulators. (The simulator themselves are complied with the host platform's native compiler, for example ANSI C). To compile own *benchmark*, its need to install GCC and GNU binutils. FORTRON and C benchmarks source can be compiled in the following way using the SimpleScalar tool set [24]:

- Compiling a C program, e.g.,

  SS<endian> -na – sstrix –gcc –o <output> <file.C> -lm

- Compiling a FORTRON program, e.g.,

  SS<endian> -na – sstrix –f77 –o <output> <file.f> -lm

- Compiling a SS assembly program, e.g.,

  SS<endian> -na – sstrix –gas –o <output> <file.f> -lm

- Disassembling a program, e.g.,

  SS<endian> -na – sstrix –objdump –{s, d} <input>

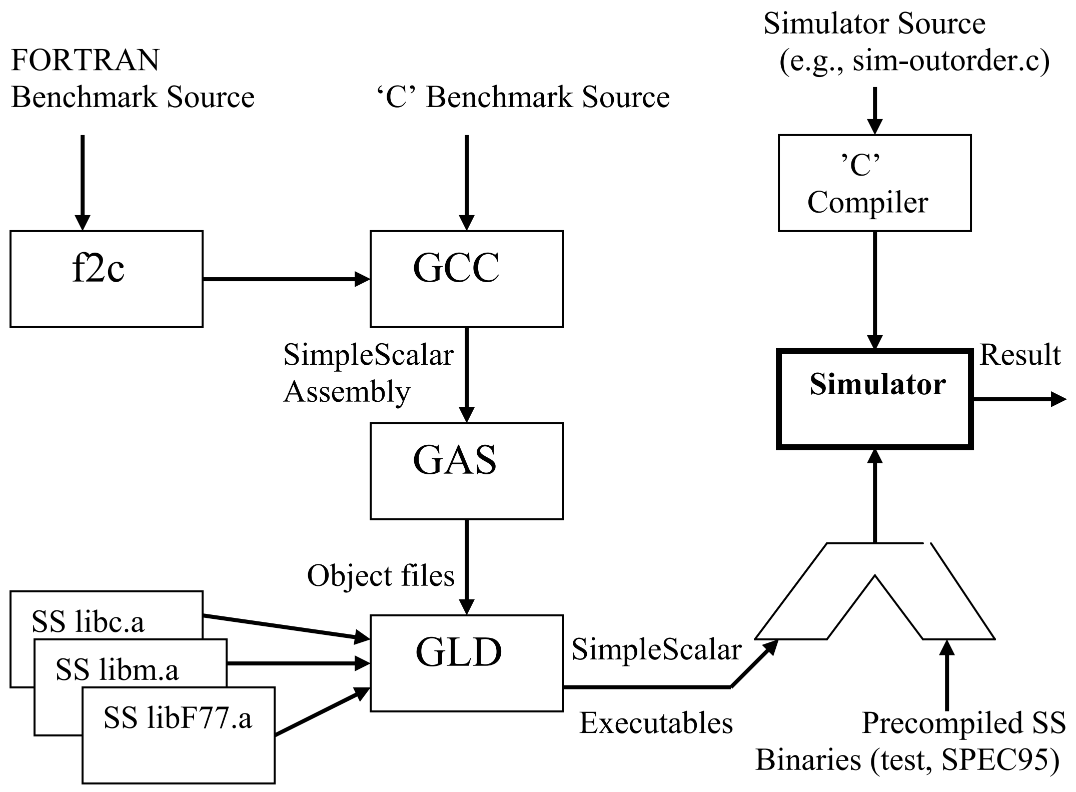


Figure 12: SimpleScalar tool set for overview

The internal simulators available in the SimpleScalar are categorized in following two sections depending upon their functionality [24]:

i) Execution and trace driven simulators

• sim-fast	functional simulation
• sim-safe	sim-fast with error detection
• sim-profile	program profiling tool
• sim-cache	functional cache simulator
• sim-cheetah	cache simulator (multiple configurations)
• sim-outorder	performance out-of-order execution

ii) Trace generator

• sim-eio	i/o-tracing & checkpoint

The following command-line arguments are available in the simulator to perform simulations as well user can configure the command line:

- Running a simulator

  sim-any { -options } executable {arguments}

- General configuration (options)

-config	<string>	-run with own configuration
-dumpconfig	<string>	-write configuration to file
-h	<true/false>	-help message
-v	<true/false>	-verbose operation
-d	<true/false>	-enable debug messages
-i	<true/false>	-start Dlite (debugger)
-seed	<int>	-random generator
-q	<true/false>	-quit immediately

- Input and output
  - Program/Trace

    ---Test-fmath, floating point test program
  - Configuration

    --- Simulator dependent
  - Program output and performance (Figure 13 shows the graphical overview at the simulator with respect to Input/output and configuration)

Out of all the simulators, those mentioned in the section Execution and trace driven simulators, the most complicated and detailed is sim-outorder [14]. This simulator supports out-of-order issues and execution, based on the Register Update unit (RUU). The *RUU* scheme uses a reorder buffer to automatically rename registers and hold the results of pending instructions. It supports *configurable architecture* and generates timing statistics of execution during simulation. It also makes use caches and branch prediction.

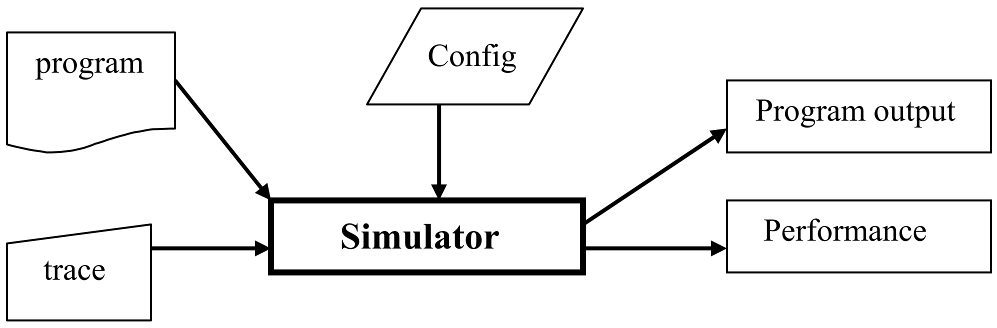


Figure 13: Input/Output sketch for Simulator

Figure 14 depicts the simulated pipeline of sim-outorder.

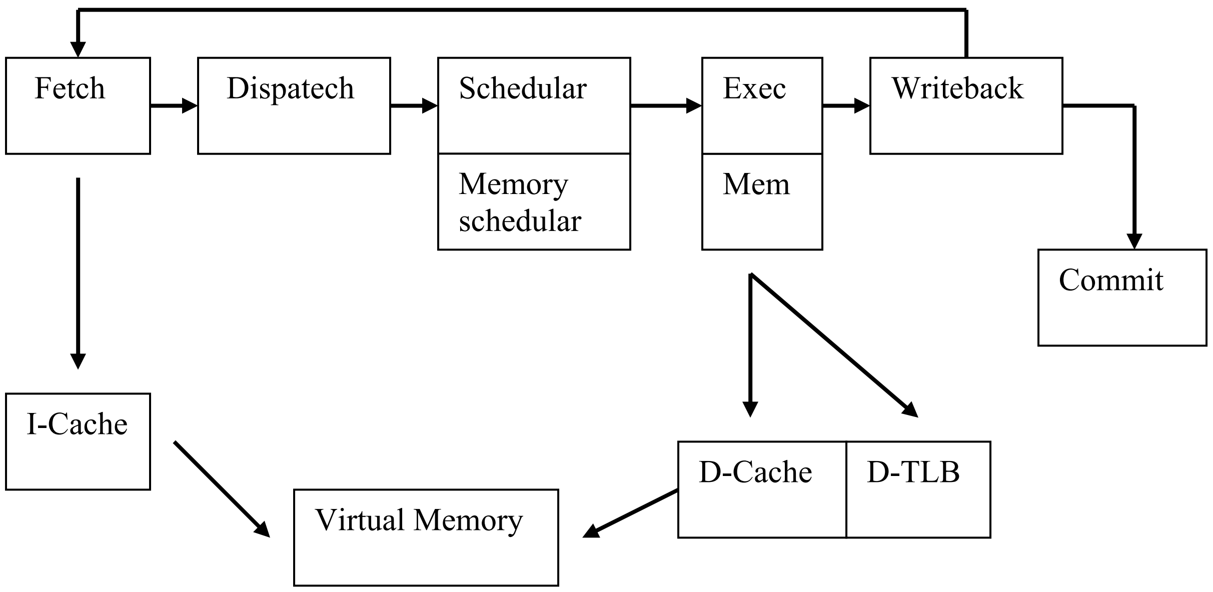


Figure 14: Pipeline for sim-outorder simulator of SimpleScalar

The key features of simulated system are listed below:

- Multiple instruction issue
- Out-of-order scheduling
- Branch prediction
- Blocking and Non-blocking loads and stores

- Optimized memory consistency implementation
- Relevant cache hierarchy
- Multiple outstanding requests
- CC-NUMA shared-memory system
- Snoop-based or directory-based cache coherence protocol with fine-grain synchronization
- Bus network for Snoop-based system or Mesh Network for Directory-based system

6.3. Simulation procedure

Suppose MICCG3D application source file is available for the simulation. There are two related benchmarks source files (new_vector.c and new_micBarrier.c) available for this application. Following is the procedure to carry out simulation:

1. Compile the source file using GCC compiler to generate assembly files

   gcc –S new_micBarrier.c

   gcc –S new_vector.c

2. Now we have the assembly files, then we need to modify the assembly files because of the way modified *LOAD* and *STORE* instruction in source file. From the assembly file these modified *LOAD/STORE* instructions can be easily find out and mark them to convey the message to SimpleScalar that these are special synchronized *LOAD/STORE* instructions
3. The marking instructions are chosen (Modified simulator is configured to recognize these marker instructions) as follows:

   xor $31, 1, $31 for LOAD

   xor $31, 2, $31 for STORE

   These marker instructions do nothing but just to indicate to the simulator.
4. Now we have modified assembly files and can generate to binary compiled file using the SMP (synchronizing memory protocol) file *new_libssmp.s* as follows:

   gcc –static –o new_miccg3d new_micBarrier.s new_vector.s new_libssmp.s –lm

5. Now we got binary compiled file *new_miccg3d,* compile and simulate the binary file as follows to get the results:

   ./ simWhatever new_miccg3d {-options}

   In the options field, we can provide number of processors and X,Y,Z dimensions used in this application. Option can be changed as per given in section 6.2, 'General preferences (options)'. For command line help, run the following instruction:

   ./simWhatever new_miccg3d –h

6. Finally, we have the result from the simulator of the proposed architecture and these results can be compared with the existing results of the *Alewife Machine* to notice the performance of the modified architecture and coherence protocol.

6.4. Simulation experiments

Initial simulations have been done by our collaborators at the University of Massachusetts (UMASS), USA using MICCG3D application in evaluation experiments [26]. They have done the modification for one of the synchronization primitives supported by the *Alewife Machine*, namely, *J-structure*. The simulation experiments have also been done at KTH, Sweden and results are reproduced within this master thesis project for the same *J-structure*.

Modification supports the management of FE-bit along with Pending bit for synchronized read miss in the cache, and it has been combined with directory-based protocol. The directory has also been modified to support the modification done in cache, SMB has been maintained to keep list of pending-read miss. To support all these modification in the structure and protocol, simulator has also been modified.

They simulated the application on the base simulator, software managed (SW) and those corresponding to their new proposed synchronization coherence protocol (SC). As the multiprocessor size is increased, and the data size is decreased, performance gets larger. It shows upto 15% performance improvement, 15% fewer network messages, and 30% fewer cache accesses [26].

More precisely in author's research work, proposed architecture also supports the pending bit for synchronized write miss and directory modification with pending-write bit. This work also supports the integration of proposed architecture with directory-based and snoop-based coherence protocol. It is need to modify the SimpleScalar [23] to support all the modification suggested in this work and then perform the simulation using simulator. From the work done at UMASS and benefits of hardware approach, it is expected to get improved performance from proposed architecture.

6.5 Application: MICCG3D

MICCG3D is the preconditioned *conjugate gradient (CG)* method knows as *Modified Incomplete Cholesky Factorization Conjugate* in 3-Dimensions [32]. The Conjugate Gradient algorithm is a semi-iterative method for solving a system of linear equations expressed in the form of matrix as Ax = b. The rate of convergence of CG method can be improved by preconditioning the system equation with a matrix K^{-1} and then applying the CG method to the preconditioned system.

The idea of using the preconditioner such that $K^{-1} A$ is close to identity matrix I [32]. Mathematically it can be expressed as follows:

$$Ax = b \qquad (1)$$

$$K^{-1} A\, x = K^{-1} b \qquad (2)$$

$$w = K^{-1} b - K^{-1} A\, x \qquad (3)\ \text{(Solver equation)}$$

But implementing the preconditioned techniques in the system, preconditioner steps involve recurrence relations which do not vectorize or parallelize easily. So, it addresses the issue of parallel performance in the system, which occurs due to recurrence relations in the preconditioned steps.

MICCG3D is difficult to parallelize because the recurrence relations in the solver operation impose data dependencies which are numerous and complex [32]. There are two ways to parallelizing MICCG3D. One uses coarse-grain barrier synchronization and other uses fine-grain data level synchronization.

In *coarse-grain* MICCG3D, barriers are placed in between vector operations to ensure that results are fully completed before being sued in subsequent computation. Barriers ensure the synchronization in the coarse-grain MICCG3D. For all but the solver operation this is sufficient to guarantee correctness. Dependencies arising from the recurrence relation in the solver require further use of barriers.

But on the other hand, in fine-grain MICCG3D, synchronization is done at word-level. Word-level synchronization automatically enforces the recurrence dependencies in the solver equation. In fine-grain version of solver equation, each processor can compute results as fast as possible. If a thread tries to read such a value that has not been computed yet, the semantics of data level synchronization force the execution of thread to stop and wait until the value becomes available. Therefore, processor never waits unnecessarily.

Our project collaborators at UMASS, USA have used the fine-grain version of MICCG3D from [32] to evaluate the performance of modified J-structure (one of the synchronized primitive of Alewife Machine) integrated with directory-based protocol. These evaluations/simulations have been done to analyze the performance improvement of the system which supports the tighter integration of fine-grain synchronization with cache coherence protocol.

7. Conclusions

This work has presented how the fine-grain synchronization could be incorporated in the multiprocessor systems with efficient hardware support. The implementation has been described here in the context of Alewife Machine and keeping it as a base line so that architectural modification could be compared with it. However, the implementation of fine-grain synchronization has introduced some extra complexity at both hardware and software level. It incorporates sophisticated cache controllers because of the increased complexity of FSM implemented with in it to handle the fine-grain synchronization.

A novel cache architecture that supports the fine-grain synchronization has been proposed. This architecture contains the full/empty bits along with pending bits for both synchronized read and synchronized write misses. The extra hardware required by this architecture is not increasing excessively and it can be compensated by the improved performance.

New cache coherence protocols combined with fine-grain synchronization have been discussed for the snoop-based and directory-based shared memory multiprocessors. Here the description includes all the transitions rules required to express the state of the protocol. We have given very strong attention in the handling of synchronized write miss along synchronized read miss. Optimization of protocol has also been explained in terms of merging and resuming of pending read/write operations.

Although the proposed architecture and its integration with fine-grain synchronization on both snoop-based and directory based protocols has not been modeled in full scale in this project work. Part of this project based on J-structure has been implemented in *SimpleScalar* by our collaborator [26] at UMASS to model a cc-NUMA multiprocessor with its hardware support for fine-grain synchronization. However, the simulator models only the part of synchronized memory operations needed for J-structures. The simulator should be extended and modified [16], [23]to support the complete set of FE-*memory* operations proposed in this project work and need to simulate on the SimpleScalar simulator and analyze its performance improvement and compared with *Alewife Machine.*

Evaluation experiments with the simulator using the MICCG3D application from SPLASH 2 has been initially done at UMASS. Also simulations have been done for this project at KTH and same results have been reproduced. Results show up to 15% performance improvement, 15% fewer network messages, and 30% fewer cache accesses [26].

8. Future work

Future work includes the developing of model, which will incorporate all features of the architecture proposed in this work. Sophisticated cache controller, complex FSM used in cache controller to handle the complexity increased by integrating fine-grain synchronization, optimized message passing in both snoop-based and directory-based protocol are need to model. This model is needed to simulate on the *SimpleScalar* to examine its performance benefits.

Since the available version of SimpleScalar does not incorporate the above proposed features so simulator platform should be modified so that the cost of including extra hardware and performance can be analyzed. Other important parameters to be measured are extra traffic caused by extra message passing and extra signals and saturation condition that may be different at different level of cache hierarchy.

Also, it is advisable to include optimization as much as possible in the different transition states of the protocols as some of them are already suggested in this proposed work. It can give more efficient network topology. This would definitely enable us to introduce more scalability in multiprocessor systems

References

[1] Agarwal, A.: "The MIT Alewife Machine: Architecture and Performance", 25 years of the International Symposia on Computer Architecture (selected papers), Association for Computing Machinery, August 1998, pages 103-110.

[2] Agarwal, A.; Bianchini, R.; Chaiken, D.; Chong, F.T.; Johnson, K.L.; Kranz, D.; Kubiatowicz, J.D.; Beng-Hong Lim; Mackenzie, K. and Yeung, D.: "The MIT Alewife Machine: Architecture and Performance", Laboratory for Computer Science, Massachusetts Institute of Technology, 1999.

[3] Agarwal, A.; Bianchini, R.; Chaiken, D.; Johnson, K.; Kranz, D.; Kubiatowicz, J.; Lim, B.H.; Mackenzie, K. and Yeung, D.: "The MIT Alewife Machine: Architecture and Performance", Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCAí95), June 1995, pages 2-13.

[4] Agarwal, A.; Kubiatowicz, J.D.; Kranz, D.; Lim, B.H.; Yeung, D.; DíSouza, G. and Parkin, M.: "Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors", Laboratory for Computer Science, Massachusetts Institute of Technology, 1993.

[5] Agarwal, A.; Beng-Hong Lim: "Waiting Algorithm for synchronization in large-Scale Multiprocessors", Massachussets Institute of Technology, ACM Transactions on Computer Systems, August 1993.

[6] Agarwal, A.; Nussbaum, Dan: "Scalability of Parallel Machine", Alewife Systems Memo # 9, Laboratory of Computer Science, MIT, November 1991.

[7] Agarwal, A.: "Overview of the Alewife Project", Alewife Systems Memo # 10, Laboratory of Computer Science, MIT, June 1990.

[8] Arvind, Rishiyur S. Nikhil: "I – Structure: Data Structure for Parallel Computing", ACM transaction on programming languages and Systems, October 1989.

[9] Chaiken, D.; Kubiatowicz, J.; Agarwal, A.: "LimitLESS Directories: A Scalable Cache Coherence Scheme", Laboratory for Computer Science, Massachusetts Institute of Technology, 1991.

[10] Chaiken, David: "Cache Coherence Protocol Specification", Alewife Systems Memo #5, Laboratory of Computer Science, MIT, April 1990.

[11] C. Scheurich, M. Dubois: "Lookup-free Caches in High-Performance Multiprocessors", Journal of Parallel and Distributed Computing 11, 25 – 36 (1991)

[12] David E. Culler, Jaswinder Pal Singh: "Parallel Computer Architecture: A hardware/software approach", Morgan Kaufmann
Publishers, 1999

[13] David E. Culler, William J. Dally, Ellen Spertus, Seth Copen Goldstein, Klaus Erik Schauser: "Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM – 5", IEEE proceeding 1993.

[14] Doug Burger, Todd M. Austin: "The SimpleScalar Tool Set, Version 2.0"

[15] Farnaz Mounes – Toussi, David J. Lilija: "Write Buffer Design for cache-Coherent Shared-Memory Multiprocessor", International Conference on Computer design, October 1995.

[16] James laudon, Daniel Lenoski: "The SGI origin: A cc-NUMA Highly Scalable Server", ACM Proceeding (ICSA 97).

[17] Johnson, K.: "Semi-C Reference Manual", Alewife Systems Memo #20, MIT Laboratory for Computer Science, version 0.6, Feb. 1992

[18] Jump, J. R.: "YACSIM Reference Manual", Rice University Electrical and Computer Engineering Department, March 1993. Available at http://www-ece.rice.edu/~rsim/rppt.html (accessed November 2001)

[19] Kourosh Gharachorlo, Anoop Gupta, John Henessy: "Performance Evaluation of Memory Consistency Models for Shared Memory Multiprocessors", ACM, 1991.

[20] Kranz, D.; Lim, B.H.; Agarwal, A. and Yeung, D.: "Low-cost Support for Fine-Grain Synchronization in Multiprocessors", in Multithreaded Computer Architecture: A Summary of the State of the Art, Kluwer Academic Publishers, 1994, pages 139ñ166

[21] Kroft, D.: "Lockup-Free Instruction Fetch/Prefetch Cache Organization", 25 years of the International Symposia on Computer Architecture (selected papers), Association for Computing Machinery, August 1998, pages 20-21

[22] Kubiatowicz, J.: "Users Manual for the Alewife 1000 Controller", Alewife Systems Memo #19, MIT Laboratory for Computer Science, version 0.69, Dec. 1991

[23] Manjikian, Naraig: “Multiprocessor Enhancements of the SimpleScalar Tool Set”, Department of Electrical and Computer Engineering, Queen’s University, Canada.

[24] Mikael Collin, Malden Nikitovic: “SimpleScalar: A flexible tool for architectural research”, Power Point Presentation at Royal Institute of Technology (KTH), Sweden.

[25] Norman P.: “WRL Research Report 91/12: Cache Write Policies”, December 1991.

[26] Raksit Ashok, Csaba Andras Moritz: “Synchronization Coherence Protocol: Unifying Synchronization and Caching in Multiprocessors”, Department of Electrical and Computer Engineering, UMASS, Amherst, MA, USA.

[27] Oscar Sierra Merino, Vladimir Vlassov, Csaba Andras Moritz: “Performance Implication of Fine-Grained Synchronization in Multiprocessors”, Master Thesis Report, Royal Institute of Technology (KTH), Sweden.

[28] Per Stenstrom, Fredrik Dahlgren, Lars Lundberg: “A Lookup-free Multiprocessor Cache Design, Department of Computer Engineering, Lund University, Sweden.

[29] Vlassov, V. and Moritz, C.A.: “Efficient Fine Grained Synchronization Support Using Full/Empty Tagged Shared Memory and Cache Coherency”, Technical Report TRITA-IT-R 00:04, Dept. of Teleinformatics, Royal Inst. of Technology, Dec. 2000

[30] Weaver, D.L. and Germond, T.: “The SPARC Architecture Manual”, PTR Prentice Hall, version 9, 1994

[31] Xiaowei, Shen and Boon, S. Ang: "Implementing I-structures at Cache Level Coherence Level", MIT Laboratory for Computer Science, 1995

[32] Yeung, D. and Agarwal, A.: “Experience with Fine-Grain Synchronization in MIMD Machines for Preconditioned Conjugate Gradient”, Principles and Practice of Parallel Programming, 1993, pages 187-197

[33] http:// www.simplescalar.com

Abbreviations

ALU	Arithmetic Logic Unit
CA	Communication Assistant
cc-NUMA	cache coherent-Non Uniform Memory Access
CMMU	Communication and Memory Management Unit
CPU	Central Processing Unit
DM	Data Memory
FSM	Finite State Machine
SM	State Memory
SMM	State Miss Memory
FE-bit	Full/Empty-bit
FGS	Fine-Grain Synchronization
MSHR	Miss-State Holding Register
Pw-bit	Pending-write bit
Pr-bit	Pending-read bit
SMP	Shared-Memory Multiprocessor
SMB	Synchronizing Miss Buffer
SPLASH 2	Stanford Parallel Applications for Shared Memory