Performance Implication of Fine-Grained Synchronization in Multiprocessors

Oscar Sierra Merino,
Vladimir Vlassov,
with Csaba Andras Moritz

Department of Microelectronics and Information Technology
Royal Institute of Technology

Stockholm 2002

KUNGL TEKNISKA HÖGSKOLAN
Royal Institute of Technology

Performance Implication of Fine-Grained Synchronization in Multiprocessors

Oscar Sierra Merino*,
Vladimir Vlassov*,
with Csaba Andras Moritz**

* Department of Microelectronics and Information Technology (IMIT)
Royal Institute of Technology (KTH), Stockholm, Sweden

** Department of Electrical and Computer Engineering
University of Massachusetts (UMASS), Amherst, MA, U.S.A.

TRITA–IMIT–LECS R 02:02
ISSN 1651–4661
ISRN KTH/IMIT/LECS/R-02/02—SE

Department of Microelectronics and Information Technology
Royal Institute of Technology
Stockholm, Sweden

Abstract

It has been already verified that hardware-supported fine-grain synchronization provides a significant performance improvement over coarse-grained synchronization mechanisms, such as barriers. Support for fine-grain synchronization on individual data items becomes notably important in order to implement thread-level parallelism more efficiently.

One of the major goals of this research is to evaluate a new efficient way to support fine-grain synchronization mechanisms in multiprocessors. This novel approach is based on the efficient combination of fine-grain synchronization with cache coherence and instruction level parallelism in a multiprocessor with the full/empty tagged shared memory. Both snoopy and directory-based cache coherence protocols have been considered.

First, we define the complete set of synchronizing memory instructions as well as the architecture of the full/empty tagged shared memory that provides support for these operations. Next, we develop a snoopy cache coherency protocol for a SMP with the centralized full/empty tagged memory, and a directory-based cache coherency protocol for a multiprocessor with the distributed full/empty tagged shared memory. A simulation environment based on the existing execution-driven simulator RSIM is designed for verification and performance evaluation of the proposed solutions. Finally, we present results of simulation experiments.

Keywords: fine-grain synchronization, shared memory, instruction-level parallelism, cache coherence, execution-driven simulation.

Table of Contents

1. Introduction 5
1.1. Related Work 5
1.1.1. The Alewife Machine 5
1.1.2. The StarT-NG machine 6
2. Architectural support for fine-grain synchronization 8
2.1. Semantics of Synchronizing Memory Operations 8
2.2. Proposed Architecture 10
2.3. Cache Coherence with Support for Fine-Grain Synchronization 12
2.4. Integration of Fine-Grain Synchronization with A Snoopy Cache Coherency Protocol 13
2.4.1. The Illinois MESI Protocol 13
2.4.2. The FE-MESI Protocol 13
2.4.3. Correspondence between processor instructions and bus transactions 16
2.4.4. Management of pending requests 17
2.4.5. State transition rules in the FE-MESI protocol 18
2.4.6. Summary 21
2.5. Integration of Fine-Grain Synchronization with A Directory-Based Protocol 22
2.5.1. Mapping between processor instructions and network transactions 24
2.5.2. Management of pending requests 25
2.5.3. Directory transition rules 27
2.5.4. Summary 34
3. Simulation 34
3.1. Features of the simulated platform 34
3.2. Preparing binary code for simulation 35
3.3. Implementation of synchronizing instructions 36
3.4. Simulation flowchart 38
3.5. Simulation results 40
. Conclusions 42
. Future work 42
. Acknowledgments 42
A. Preparing binaries for simulation 43
B. Application source (fine-grained version) 45
C. Application source (coarse-grained version) 47
. References 49

1 Introduction

There are two general types of synchronization that guarantee correctness of execution in shared-memory programming model: mutual exclusion and condition synchronization. With mutual exclusion, only one process (thread) may execute its critical session at a time, whereas with condition synchronization a process may be suspended until some certain condition is met. There exist several synchronization mechanisms that allow to achieve mutual exclusion or condition synchronization, such as locks and barriers.

Barriers are an example of synchronization that ensure the correctness of a producer-consumer behavior. They are coarse-grain in the sense that all processes participating in a barrier have to wait in a common point (i.e. at the barrier), even though the data a barriered process truly depends on can be already available.

The main advantage of fine-grain synchronization arises from the fact that synchronization is provided at data-level. As a consequence, false data dependencies and unnecessary delays caused by the coarse-grained synchronization such as barrier can be avoided. Communication overhead due to global barriers is also avoided, because each process communicates only with the processes it depends on. Thus, the serialization of program execution is notably reduced and more parallelism can be exploited. This effect is more noteworthy as the number of processors increases. While the overhead of a fine-grain synchronization operation remains constant, that of a coarse-grain operation typically increases with the number of processors.

As explained in [24], fine-grain synchronization is most commonly provided by three different mechanisms:

- language-level support for expressing data-level synchronization operations,
- full/empty bits storing the synchronization state of each memory word,
- processor operations on full/empty bits.

Traditional theory on data-level parallelism has led to the definition of specific structures supporting fine-grain synchronization in data arrays such as write-once I-structures and M-structures. As another example, J-structures provide consumer-producer style of synchronization, while L-structures guarantee mutual exclusion access to a data element [3]. Both data types associate a state bit with each element of an array.

Several alternatives exist for handling a synchronization failure. The most immediate are either polling the memory location until the synchronization condition is met or blocking the thread and returning the control at a later stage, which requires more support as it is necessary to save and restore context information. A combination of both is another option, polling first for a given period and then blocking the thread. The waiting algorithm may depend on the type of synchronization being executed [18].

Most research regarding multiprocessors show that fine-grain synchronization is a valuable alternative for improving the performance of many applications. As exposed in [15], evidence is shown on the worthiness of having hardware support for fine-grain synchronization. Testing the benefits of aggressive hardware support in fine-grain synchronization is one of the goals of this research project.
Before specifying an architecture of the full-empty tagged memory, let us give a short overview of related work.

1.1 Related Work

1.1.1 The Alewife Machine

The MIT Alewife machine is a cache-coherent shared memory multiprocessor (see [1] and [3]) with non-uniform memory access (NUMA). Although it is internally implemented with an 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 protocols is the communication and memory management unit (CMMU), which deals with memory requests from the processor and determines whether a remote access is needed, managing also cache fills and replacements. Cache coherency is achieved through LimitLESS, a software extended directory-based protocol. The home node of a memory line is responsible for the coordination of all coherence operations for that line.

Support for fine-grain synchronization in Alewife includes full/empty bits for each 32-bit data word and fast user-level messages. Colored load and store instructions are used to access synchronization bits. The alternate space indicator (ASI) distinguishes each of these instructions. Full/empty bits are stored in the bottom four bits of the coherency directory entry (at the memory) and as an extra field in the cache tags (at the cache), so they do not affect DRAM architecture nor network data widths. The Alewife architecture also defines language extensions to support both J- and L-structures. A specific programming language, namely Semi-C[1], has been defined for this purpose [13].

The aim is that a successful synchronization operation does not incur much overhead with respect to a normal load or store. In the ideal case, the cost of both types of operations is expected to be the same. This is possible because full/empty bits can be accessed simultaneously with the data they refer to. The cost of a failed synchronization operation depends much on the specific hardware support for synchronization. The overhead of software-supported synchronization opera-

1. Semi-C is an extension of the C language that can handle parallel programming constructs.

tions is expected to be much higher than their hardware counterparts. However, Alewife minimizes this by rapidly switching between threads on a failed synchronization attempt or a cache miss, requiring the use of lockup-free caches.

Handling failed synchronization operations in software has the advantage of being less complex in terms of hardware and more flexible. The basis of Alewife support for fine-grain synchronization is that, as synchronization operations are most probably successful, overhead due to such failures is not expected to notably reduce overall system performance.

1.1.2 The StarT-NG machine

StarT-NG, an improved version of the StarT machine [6], is a high-performance message passing architecture in which each node consists of a commercial symmetric multiprocessor (SMP) that can be configured with up to 3 processors, which are connected to the main memory by a data crossbar. At least one network interface unit is present in each node, allowing communicating with a network router, which is implemented in a proprietary chip [7].

A low-latency high-bandwidth network interconnects every node in the system. StarT-NG also supports cache-coherent global shared memory. In this case, one processor on each site is used to implement the shared memory model. This functionality can be disabled when shared memory is not needed.

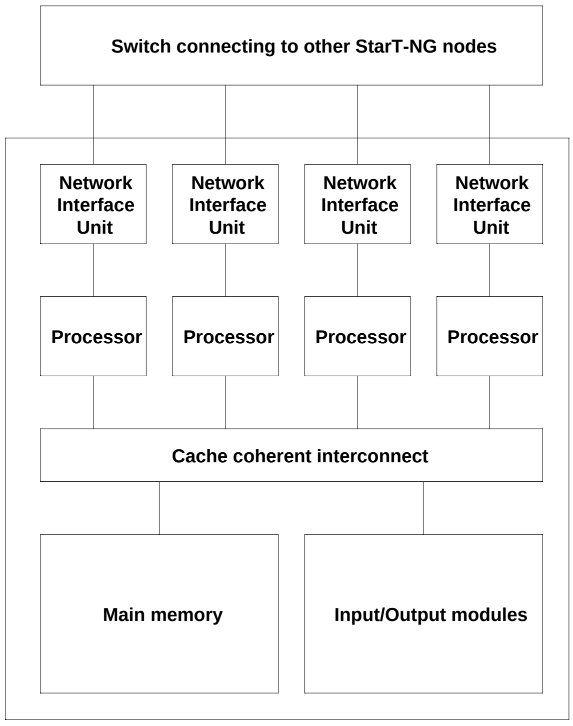
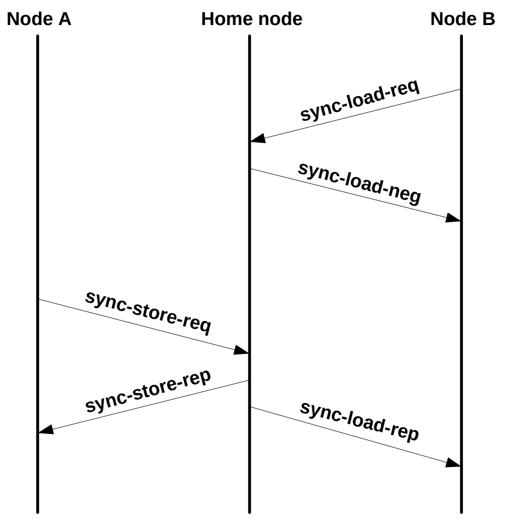


Figure 1: Architecture of a StarT-NG node [9]

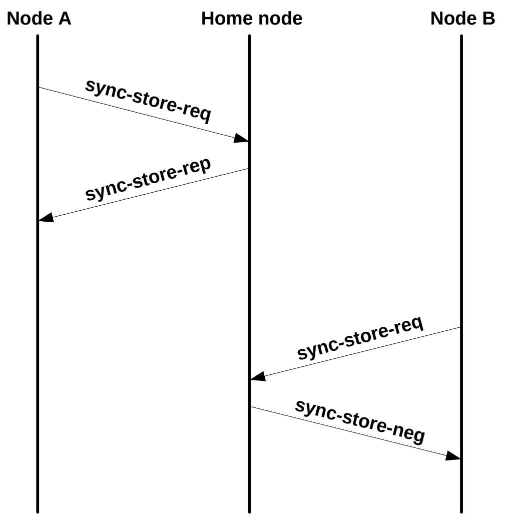
Coherence protocols in StarT-NG are fully implemented in software. As a consequence, the choice of protocols and configuration of the shared memory is notably flexible. The performance of several coherence models has been evaluated. Particularly relevant to this work is the study made in [23], which introduces a cache coherence protocol with support for fine-grained data structures. These data structures are known as I-structures [23].

According to the results of this study, performance improvements in an integrated coherence protocol are two-fold. First, the write-once behavior of I-structures allows writes to be performed without the exclusive ownership of the respective cache line. Once a write has been carried out, stale data in other caches is identified because its full/empty bit is unset. In a directory-based protocol, a synchronized load in a remote location will find the full/empty bit unset and forward the request to the proper node. This behavior is illustrated in Figure 2, where two nodes (namely, A and B) share a copy of a block on which they perform different operations.

As stated in [22], another advantage of a coherence protocol integrated with fine-grain synchronization is the efficiency in the management of pending requests by reducing the number of transactions needed to perform some particular operations. As an example, a synchronized load in traditional coherence protocols usually requires the requesting node to obtain the exclusive ownership of the affected block in order to set the full/empty bit to the empty state.



Scenario 1 Initially, both nodes A and B have a copy of the cache line in the shared state. A synchronized store operation is performed by node A without the *exclusive ownership* of the cache block, which is consequently kept in the *shared* state during the whole process. Pending synchronized loads from node B to the affected slot are resumed after the store is performed.



Scenario 2 Initially, both nodes A and B have a copy of the cache line in the shared state. A synchronized store operation is successfuly performed by node A without the *exclusive ownership* of the cache block. If node B issues a synchronized store, the request will be rejected by the home node after finding the *full-empty* bit set.

Figure 2: Two sample scenarios of synchronized loads and stores

2 Architectural support for fine-grain synchronization

2.1 Semantics of Synchronizing Memory Operations

Synchronization operations require the use of a tagged memory, in which each location is associated to a state bit in addition to a value stored in the location. The state bit is known as full/empty bit, shortly *FE-bit*, and it implements the semantics of synchronizing memory accesses. As a matter of fact, this bit controls the behavior of synchronized loads and stores. For example, a set FE-bit indicates that the corresponding memory reference has been written by a successful synchronized store. On the contrary, an unset FE-bit means either that the memory location has never been written since it was initialized or that a synchronized load has read it.

The full/empty-tagged memory, shortly *FE-memory*, is the memory in which each word has a FE-bit associated with it. In general., the FE-memory can be composed of two parts: (1) the data memory which holds data, and (2) the state memory which holds FE-bits. A memory operation on the FE-memory can access either of these parts or both. The joint diagram depicted in Figure 3 shows possible combinations of read (Rd) or write (Wr) operations that access the data memory with operations set-to-Empty (E) and set-to-Full (F) that access the state part of the memory. Combined operations such as Rd&E (read and set to Empty) and Wr&F (write and set to Full) are atomic.

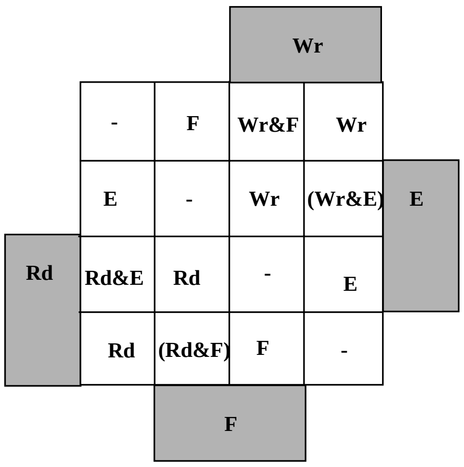


Figure 3: Possible combinations of operations on the data memory (Rd: read, Wr: write) and operations on the FE-bits (E: set to Empty, F: set to Full). Combined operations such as Rd&E (read and set to Empty) are atomic.

We also define conditional memory operations that depend on the FE-state of the target location.

A categorization of the different synchronizing memory operations as proposed earlier in [20] is depicted in Figure 4. These instructions are introduced as an extension of the instruction set of Sparcle [5], which is in turn based on SPARC [21]. The simplest type of operations includes *unconditional* (ordinary) `load` and store, setting and resetting the full/empty bit or a combination of these. As they do not depend on the previous value of the full/empty bit, unconditional operations always succeed.

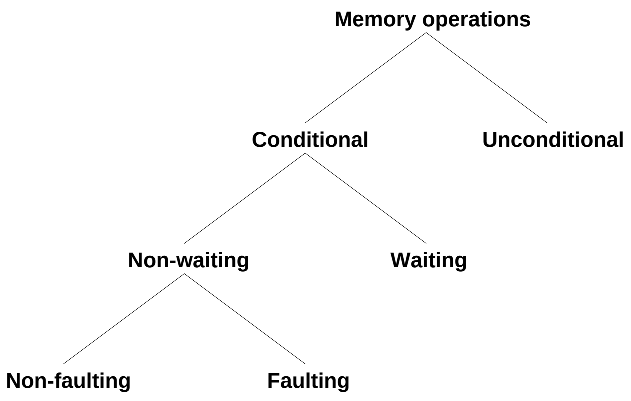


Figure 4: Categorization of the FE-memory operations

Conditional operations depend on the value of the full/empty state bit to successfully complete. A conditional read, for instance, is only performed if the state bit of the location being accessed it set. The complimentary applies for a conditional write. Conditional memory operations can be either *waiting* or *non-waiting*. In the former case, the operation remains pending in the memory until the state miss is resolved. This introduces non-deterministic latencies in the execution of synchronizing memory operations. Lastly, conditional non-waiting operations can be either *faulting* or n*on-faulting*. While the latter do not treat the miss as an error, faulting operations fire a trap on a state miss and either retry the operation immediately or switch to another context.

All memory operations, regardless of the classification made in Figure 4, can be further catalogued into *altering* and *non-altering* operations. While the former modify the full/empty bit after a successful synchronizing event, the latter do not touch this bit in any case. According to this distinction, ordinary memory operations fall into the *unconditional non-altering* category.

Table 1 shows the notation used for each variant of memory operations and its behavior in the case of a synchronization miss. The notation is further explained in Figure 5.

Table 1: Notation of FE-memory operations

Notation	Semantics	Behavior on a synchronization 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	Placed on the list of pending requests until resolved
`WNWr`	Waiting and non-altering write to empty	
`WARd`	Waiting and altering read from full	
`WAWr`	Waiting and altering write to empty	
`NNRd`	Non-faulting and non-altering read from full	Silently discarded
`NNWr`	Non-faulting and non-altering write to empty	
`NARd`	Non-faulting and altering read from full	
`NAWr`	Non-faulting and altering write to empty	
`TNRd`	Faulting and non-altering read from full	Signal trap
`TNWr`	Faulting and non-altering write to empty	
`TARd`	Faulting and altering read from full	
`TAWr`	Faulting and altering write from empty	

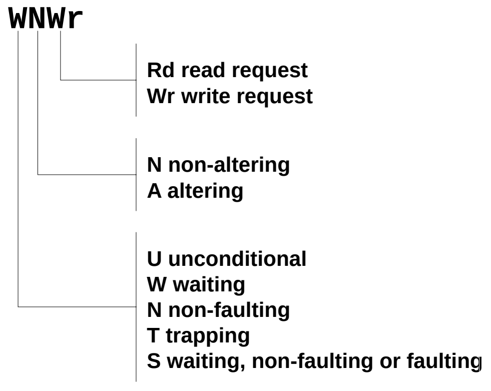


Figure 5: Notation of FE-memory operations

2.2 Proposed Architecture

In a multiprocessor system providing fine-grain synchronization, each shared memory word is tagged with a full/empty bit that indicates the synchronization state of the referred memory location. Assuming that a memory word is 32-bit long, this implies an overhead of just 3%. Although many variations exist when implementing this in hardware, the structure of shared memory is conceptually as shown in Figure 6.

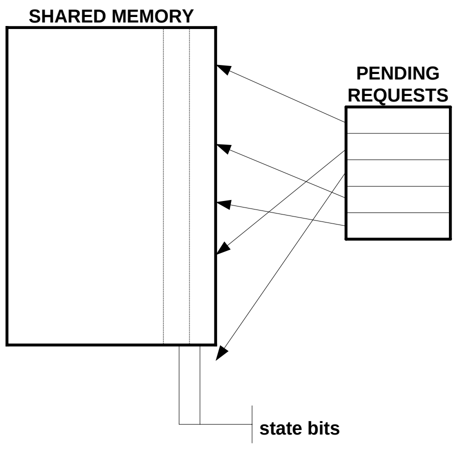


Figure 6: Logical structure of the Full-Empty tagged shared memory.

Figure 6 shows that each shared memory location (a word) has three logical parts, namely:

- The shared data itself.
- State bits associated with the location. The full/empty bit is placed within the state bits. This bit is set to 1 if the corresponding memory location has already been written by a processor and thus contains valid data. If the architecture has cache support other state bits such as the dirty bit may exist. The dirty bit is set if the memory location is not up-to-date, indicating that it has been modified in a remote node.
- The list of pending memory requests. Synchronization misses fired by conditional waiting memory operations are placed in this list. When an appropriate synchronizing operation is performed, the relevant pending requests stored in this list are resumed. If the architecture has cache support, the list of pending memory requests also stores ordinary cache misses. The difference between both types of misses is basically that synchronization misses store additional information, such as the accessed slot is index in the corresponding cache block. These differences are further explained later in this section.

Note that fine-grain synchronization is described here only for shared memory locations. In the presented architecture, the local memory in each processing node does not make use of full/empty bits. With this consideration, the memory map of the system seen by each processor is similar to the one sketched in Figure 7.

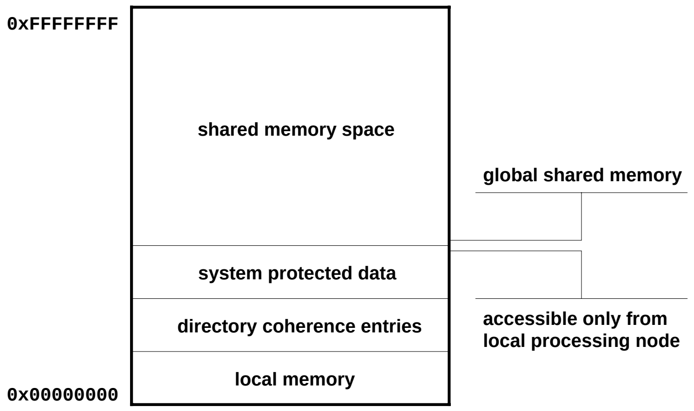


Figure 7: Memory map for each processing node

Fine-grain synchronization is implemented by atomic test-and-set operations. These operations modify the full/empty condition bit in the processor's condition bits register[1]. Note that the condition bit is changed regardless of the particular variant of synchronization operation; no matter it is altering and/or trapping.

As stated before, many implementation alternatives are possible. State bits may be stored in the coherence directory entry in the case of a directory-based protocol, such as the one implemented in the MIT Alewife multiprocessor [4]. A structure for a cache supporting fine-grain synchronization proposed in this report is depicted in Figure 8.

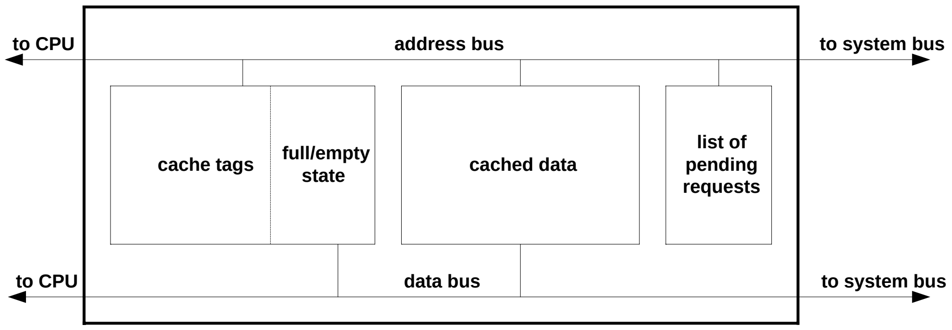


Figure 8: Organization of a cache supporting fine-grain synchronization

When a memory word is cached, its full/empty bit must also be stored at the cache side. As a consequence, not only data has to be kept coherent, but also full/empty bits. In a system with cache support, an efficient option is to store the full/empty bit as an extra field in the cache tag, allowing checking the synchronization state in the same step as the cache lookup. The coherence protocol has then two logical parts, one for the data and another for the synchronization bit.

Our design assumes that the smallest synchronizing element is a word. As a cache line is usually longer, it may contain multiple elements, including both synchronized and ordinary words. A tag for a cache line includes the full/empty bits for all the words that are stored in that line even though some of the FE-bits can be not in use. As directory states are maintained at cache line level, this complicates the maintenance of pending memory requests. Effectively, while a *dirty* bit refers to a complete cache line, a full/empty bit refers to a single word in a cache line.

In the proposed architecture, lists of pending requests (unresolved synchronization misses) are maintained in hardware at the cache level, more concretely in the *miss status holding registers* (MSHR). With this assumption, waiting memory operations require the architecture to have cache support. However, if cache support is not available, the behavior of waiting operations can be implemented in software by using faulting conditional operations instead. The system kernel is then responsible for maintaining the list of pending requests [11]. In the case of a directory-based coherence protocol, an alternative is to store the pending requests as a separate field in the directory entries.

Some modifications have to be made to the cache architecture in case synchronization misses are to be kept in MSHR. More concretely, MSHR in traditional lockup-free caches store the information listed in Table 2 (see [16] for a more detailed description). 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 operation that will be performed.

A complete description of a cache coherence mechanism should include the states, the transition rules, the protocol message specification and the description of cache line organization and memory management of pending requests. Other design issues to be considered are dealing with conflicting and/or merging synchronization misses, as well as ordering of misses from the same processor.

Our design is based on a multiprocessor system with the following assumptions:

- The CPU implements out-of-order execution of instructions;
- Each processing node has a miss-under-miss lockup-free cache, supporting multiple outstanding memory requests;
- The smallest synchronized data element is a word; this statement does not imply a loss of generality, as the extension of the presented design to other data sizes is straightforward.

1. In Sparcle (a processor of the MIT Alewife multiprocessor), for instance, the full/empty condition bit is stored in the condition bit #0 (see [21]).

Table 2: Relevant information stored in ordinary MSHR registers [16]

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

2.3 Cache Coherence with Support for Fine-Grain Synchronization

In a multiprocessor system, cache memory local to each processing node can be used to speed up memory operations. It is necessary to keep the caches in a state of coherence by ensuring that modifications to data that is resident in a cache are seen in the rest of the nodes that share a copy of the data. This can be achieved in several ways, which may depend on the particular system architecture. In bus-based systems, for instance, cache coherence is implemented by a 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 contrary, mesh network-based multiprocessors use a directory structure to ensure cache coherence. In these systems, each location in the shared memory is associated with a directory entry that keeps track of the caches that have a copy of the referred location. Both, *snoopy* and *directory-based* mechanisms can be further classified into write-invalidate and write-update protocols. In the former case, when a processors writes shared data in its cache, all other copies, if any, are set as invalid. Update protocols change copies in all caches to the new value instead of marking them as invalid.

The performance of multiprocessor systems is partially limited by cache misses and node interconnection traffic. Consequently, cache coherence mechanisms play an important role in solving the problems associated with shared data. Another performance issue is the overhead imposed by synchronizing data operations. In the case of systems that provide fine-grain synchronization, this overhead is due to the fact that synchronization is implemented as a separate layer over the cache coherence protocol. Indeed, bandwidth demand can be reduced if no data is sent in a synchronization miss. This behavior requires the integration of cache coherence and fine-grain synchronization mechanisms. It is important to remark, however, that both mechanisms are conceptually independent. This means that synchronizing operations can be implemented in machines without cache support and vice-versa.

One of the main objectives of this project is to define a coherence protocol that integrates fine-grain synchronization. This will be done for both snoopy and directory-based protocols. An event-driven simulator, namely RSIM, is used in order to verify and measure the performance of our design. As this simulation platform does not integrate synchronization at the cache coherence level, modifications in its source code are needed.

In the proposed architecture, failing synchronizing events are resolved in hardware. The following architecture requirements must be considered in order to integrate synchronization and cache coherency. Note that most of the hardware needed is usually already available in modern multiprocessor systems:

- Each memory word has to be associated with a full/empty bit; as in Alewife, this state information can be stored in the coherency directory entry,
- At the cache side, state information is stored as an additional field in the cache tags; a lookup-free cache is needed in order to allow non-blocking loads and stores;
- The cache controller not only has to deal with coherency misses, but also with full/empty state misses; synchronization is thus integrated with cache coherency operations, as opposed to Alewife, in which the synchronization protocol is implemented separately from the cache coherency system.

This approach can be extended to the processor registers by adding a full/empty tag to them. This would allow an efficient execution of synchronization operations from simultaneous threads on the registers. However, additional modifications are needed in the processor architecture to implement this feature.

In order to evaluate the performance improvement of this novel architecture with respect to existing approaches, appropriate workloads must be tested on the devised machine. A challenge task is to find suitable applications that show these results in a meaningful way, so that the effects of the synchronization overhead such as the cost of additional state storage, execution latency or extra network traffic can be studied in detail.

2.4 Integration of Fine-Grain Synchronization with A Snoopy Cache Coherency Protocol

We consider a bus-based system such that depicted in Figure 9. Note that even though each memory address has conceptually a list of pending operations for that address, at hardware level the lists are distributed among all the processing nodes. The management of deferred lists will be explained later in this section.

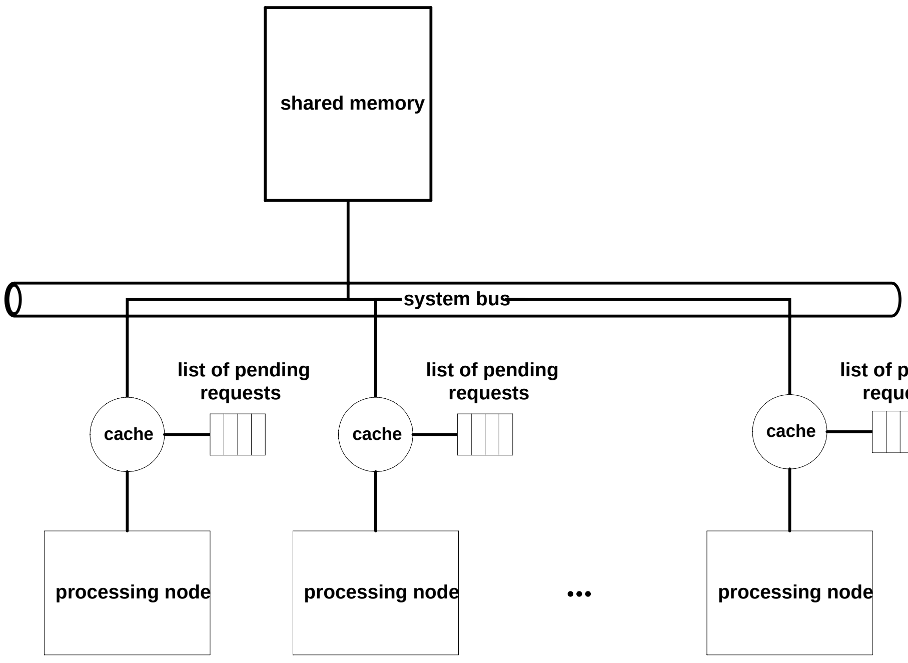


Figure 9: Bus-based multiprocessor architecture

2.4.1 The Illinois MESI Protocol

The description made here is based on the MESI protocol, also known as Illinois protocol. It is a four-state write-invalidate protocol for a write-back cache with the following state semantics [10].

- *Modified* (M) - this cache has the only valid copy of the block; the location in main memory is invalid.
- *Exclusive clean* (E) - this is the only cache that has a copy of the block; the copy in main memory is up-to-date. A signal `S` is available to the controller in order to determine on a `BusRd` if any other cache currently holds the data.
- *Shared* (S) - the block is present in an unmodified state in this cache, main memory is up-to-date and zero or more caches may also have a shared copy.
- *Invalid* (I) - the block does not have valid data.

The state diagram corresponding to the MESI protocol without fine-grain synchronization support is shown in Figure 10. In the figure, we use the notation `A/B`, where `A` indicates an observed event and `B` is an event generated as a consequence of `A` [10]. Dashed lines show state transitions due to observed bus transactions, while continuous lines indicate state transitions due to local processor actions. Finally, the notation `Flush'` means that data is supplied only by the corresponding cache. Note that this diagram does not consider transient states used for bus acquisition.

2.4.2 The FE-MESI Protocol

The transitions needed to integrate fine-grain synchronization in MESI are sketched in Figure 11, in which the full/empty state of the accessed word is explicitly indicated by splitting each state of the original MESI protocol into two states: one where FE-bit is Full and another where FE-bit is Empty. The modified MESI protocol is called FE-MESI protocol, where FE stands for Full and Empty, respectively. The transactions not shown in this figure are not relevant for the corresponding state and do not cause any transition in the receiving node. The notation is the same as in the previous figure, and as it can be appreciated below, no new states are preliminarily required so as to integrate fine-grain synchronization in the coherence protocol.

For simplicity but without loosing generality, the description made here considers only two types of FE-memory operations issued by the processor: *waiting non-altering read* (PrWNRd) and *waiting altering write* (PrWAWr), see Table 1.

As an implementation option, the altering read operation can be achieved by issuing non-altering read in combination with an operation that clears the full/empty bit without retrieving data, i.e. sets FE-bit to Empty. This operation can be named *unconditional altering clear,* or PrUACl according to the nomenclature previously described. PrUACl operates on a full/empty bit without accessing or altering the associated data.

Clearing of full/empty bits is necessary in order to reuse synchronized memory locations (a more detailed description is made in [15]). While a PrUARd could be used for this end, the PrUACl instruction completes faster, as it alters the full/ empty bit without actually reading data from the corresponding location.

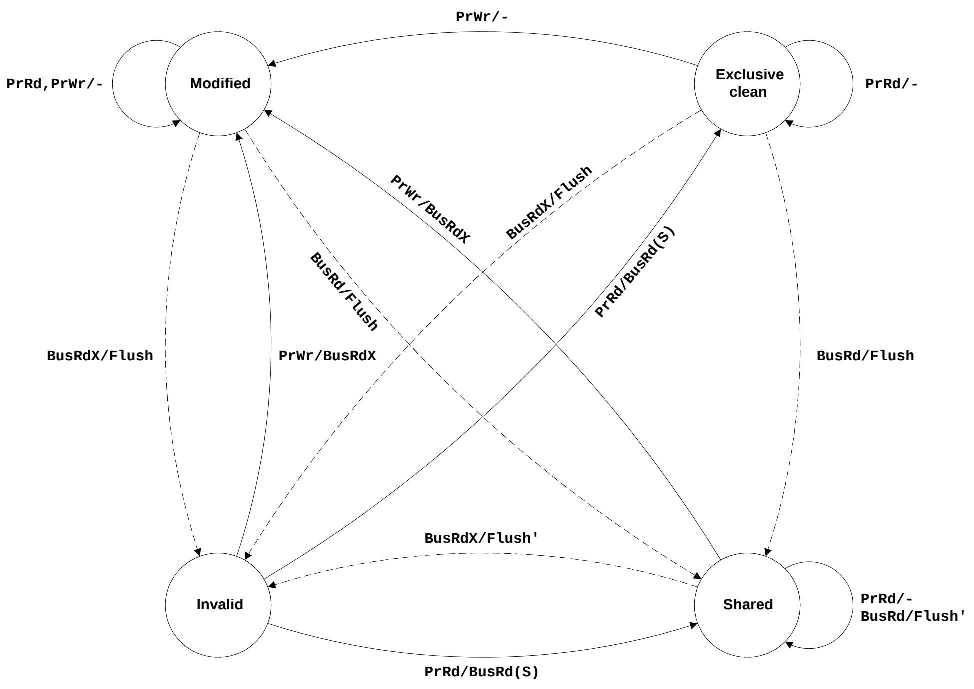


Figure 10: A state diagram for the Illinois MESI protocol

Using the operations PrWNRd (waiting non-altering reads), PrWAWr (waiting altering write) and PrUACl (unconditional altering clear), one can implement, for example, I-structures (write-once variables) and M-structures (reusable I-structures).

Waiting operations constitute the most complex sort of synchronizing operations, as they require additional hardware in order to manage deferred lists and resume pending synchronization requests. The behaviour of other types of memory operations is a simplified version of waiting operations. Most of the transitions depicted in Figure 11 are identical in the rest of the cases, with the only different being the behaviour when a synchronization miss is detected. Instead of being added to the list of pending requests, other variants of missing operations either fire an exception or are silently discarded.

Two additional bus transactions are needed in order to integrate fine-grain synchronization in the MESI protocol. A detailed description of these bus transactions is presented in Table 3. Coherence of full/empty bits is ensured precisely by these two bus transactions (BusSWr and BusSCl).

Table 3: Additional bus transactions in the FE-MESI protocol

Bus transaction	Description
`BusSWr`	A node has performed an altering waiting write. The effect of this operation in observing nodes is to set the full/empty bit of the referring memory location and resume the relevant pending requests. Resuming of pending requests is further explained in section 0.
`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 full/empty bit of the referring memory location, thus making it reusable.

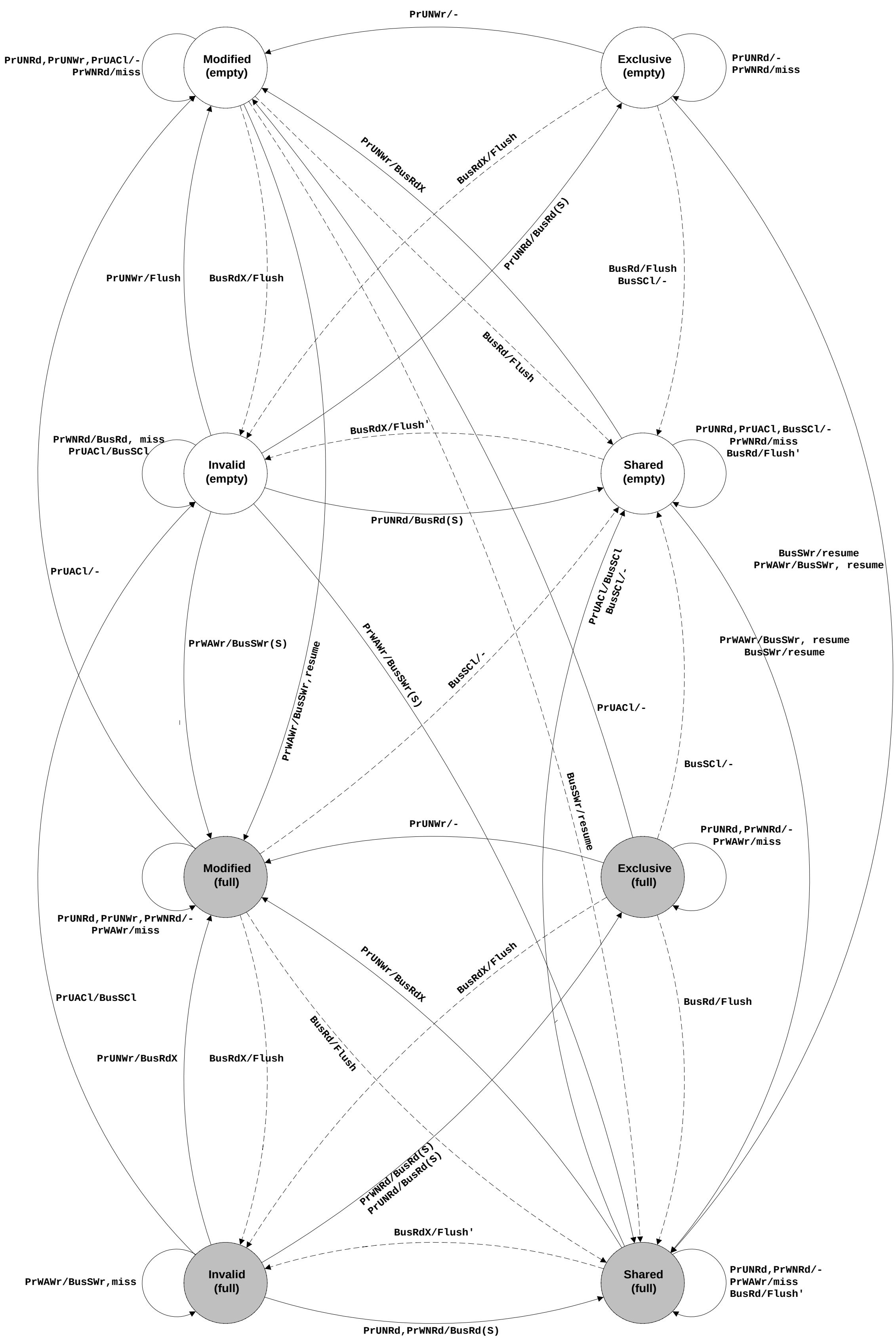


Figure 11: A state diagram of the FE-MESI protocol with explicit full/empty states

We introduce a new signal *C* in order to determine whether a synchronized operation misses. This bus signal will be called *shared-word signal*, as it indicates whether any other node is sharing the referring word. The shared-word signal can be implemented as a wired-OR controller line, which is asserted by each cache that holds a copy of the relevant word with the full/empty bit set. According to this notation, a waiting read request written in the form PrWNRd(C) successfully performs, while an event of the form PrWNRd($\overline{\text{C}}$) causes a synchronization miss. Note also that, as each cache line may contain several synchronized data words, it is necessary to specify the specific word to which the synchronized operation is to be performed. Consequently, a negated synchronization signal ($\overline{\text{C}}$) causes a requesting read to be appended to the list of pending operations whereas a requesting write to be performed successfully. If the synchronization signal is otherwise asserted (C), then a synchronized read is completed successfully whereas a requesting write is suspended.

In addition to the *shared-word* signal already introduced, three more wired-OR signals are required for the protocol to operate correctly, as described in [10]. 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 former 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 indicate whether all the caches have completed their snoop, that means, if it is reliable to read the value of the first two signals.

Figure 12 shows a more compact state transition specification in which information about the full/empty state of the accessed word is implicit. Instead, the value of the C line or the full/empty bit is specified as a required condition between parentheses. Figure 11 and Figure 12 do not consider neither transient states needed for bus acquisition nor the effects due to real signal delays.

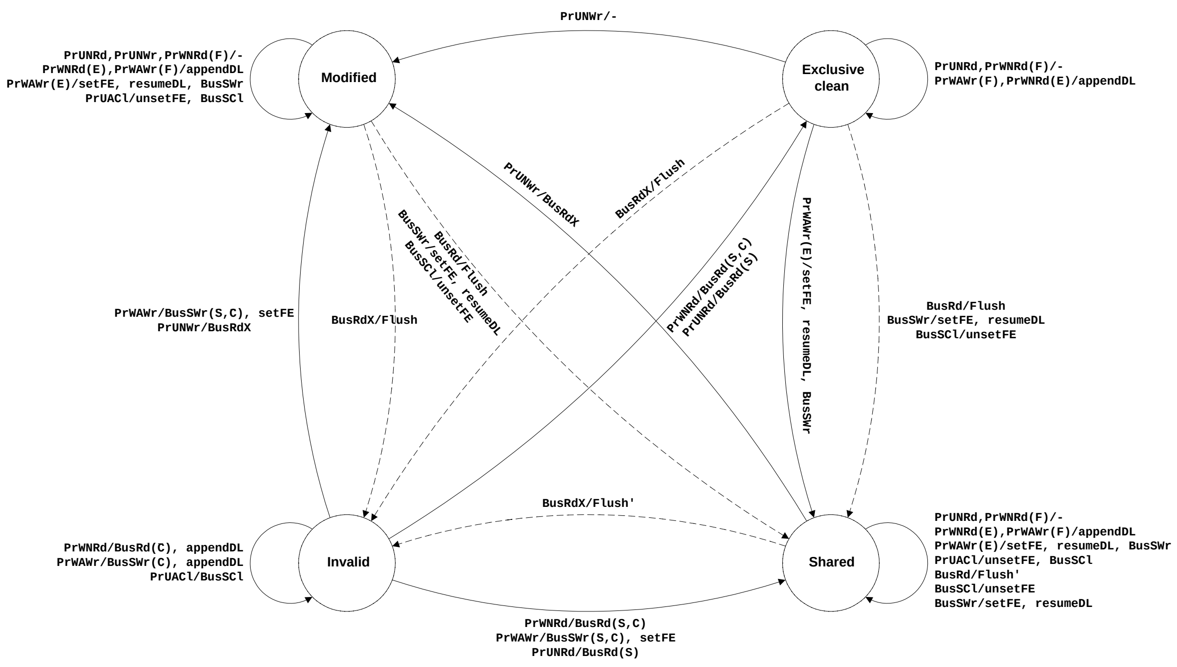


Figure 12: A state diagram of the FE-MESI protocol with implicit full/empty states.

2.4.3 Correspondence between processor instructions and bus transactions

When a processing node issues a memory operation, the cache of that node first interprets the request and, in case of a miss, it later translates the operation into one or more bus transactions. The correspondence between the different processor instructions and the memory requests seen on the bus is shown in Table 4. The same notation as in Figure 5 is used.

As seen on Table 4, unconditional non-altering read and write requests generate ordinary read and write transactions on the bus. On the contrary, an unconditional altering read requires a BusRd transaction followed by a BusSCl transaction. Effectively, apart from retrieving the data from the corresponding memory location, a PrUARd request also clears the full/empty state bit of the referring location. This is performed by BusSCl, which does not access nor modifies the data but only the full/empty bit. It is important to observe that an unconditional altering read cannot be performed by just a BusSCl transaction, as it just alters the full/empty bit without retrieving any data. The last unconditional operation, PrUAWr, generates a specific bus transaction, namely BusAWr, which unconditionally sets the full/empty bit after writing the corresponding data to the accessed memory location.

It is inferred from Table 4 that the behavior of all conditional memory operations depends on the value of the shared-word bus signal. A conditional non-altering read, for instance, generates an ordinary read bus transaction after checking whether the shared-bus signal is asserted. A conditional altering read generates a BusSCl transaction in addition to the ordinary read transaction. Finally, a conditional altering write causes a BusSWr transaction to be initiated on the bus. This transaction sets the full/empty bit after writing the corresponding data to the referred memory location.

Table 4: Correspondence between processor instructions and memory requests

Request from processor	Corresponding bus transaction issued on a miss
PrUNRd	BusRd (ordinary read)
PrUNWr	BusWr (ordinary write)
PrUARd	BusRd & BusSCl
PrUAWr	BusAWr[a]
PrSNRd	BusRd(C)[b]
PrSNWr	BusWr(C)
PrSARd	BusRd(C) & BusSCl
PrSAWr	BusSWr(C)

a. Neither unconditional altering writes nor conditional non-altering writes are considered here.
b. The bus transaction BusRd is in this case used in combination with the shared-copy signal.

Note that all synchronized operations generate the same bus transactions regardless of their particular type (waiting, non-faulting or faulting). The difference resides in the behavior when a synchronization miss is detected and not in the bus transactions issued as a consequence of the request. A sample scenario is shown in Figure 13.

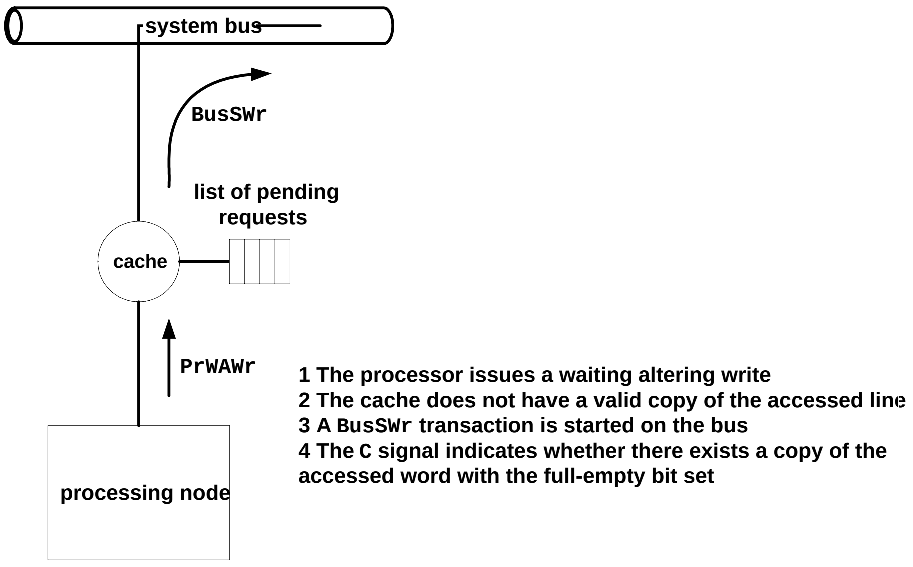

system bus
BusSWr
list of pending requests
cache
PrWAWr
processing node
1 The processor issues a waiting altering write
2 The cache does not have a valid copy of the accessed line
3 A BusSWr transaction is started on the bus
4 The C signal indicates whether there exists a copy of the accessed word with the full-empty bit set

Figure 13: Sample scenario of mapping between processor instructions and bus transactions

2.4.4 Management of pending requests

Each processing node keeps a local deferred list. This list holds both ordinary presence misses and synchronization misses. It is possible also for both types of misses to happen for a single access. In this case, not only the accessed line is not present in the cache, but also the synchronization state is not met at the remote location where the copy of the word is held. After a relevant full/empty bit change is detected, any operation that matches a required synchronization state is resumed at the appropriate processing node.

Table 5 shows how the management of the deferred list of memory operations local to a node is done. Concretely, the table specifies whether an *incoming read* request can be merged with a *pending read* request that is already on the list of deferred operations. Write requests are not shown in the table, because writes are always conflicting and cannot be merged at all. If an incoming request is conflicting with a pending request, then the request need to be kept separated into two different entries, always ensuring that local order is maintained. On the contrary, if an incoming request can be merged with a request that is already pending, then both requests are treated as a sole request from the point of view of memory accesses.

Table 5: Management of coalescing read requests

			Pending read request (already in MSHR)							
			Non-altering				Altering			
			PrUNRd	PrWNRd	PrNNRd	PrTNRd	PrUARd	PrWARd	PrNARd	PrTARd
Incoming read request	Non-altering	PrUNRd	Can be merged							
		PrWNRd	Can be merged				Conflicting			
		PrNNRd	Can be merged				Conflicting			
		PrTNRd	Can be merged				Conflicting			
	Altering	PrUARd	Can be merged							
		PrWARd	Can be merged				Conflicting			
		PrNARd	Can be merged				Conflicting			
		PrTARd	Can be merged				Conflicting			

As a rule of thumb, a pending write is conflicting with any incoming request, so it can never be merged and requires a separate entry in the list of pending requests[1]. As they are always conflicting, all write requests have been excluded from Table 5. Another important observation is that pending altering reads can only be merged with unconditional operations. Additionally, all non-altering pending reads can be coalesced with any other incoming read request.

Apart from coalescing of requests, it is also crucial to specify how resuming of pending requests is done. As explained at the beginning of this section, coherence of full/empty state bits is ensured by proper bus transactions, to be precise, BusSWr and BusSCl. This means that all caches that have pending requests for a given memory location will know when the synchronization condition is met by snooping into the bus and waiting for a BusSWr or a BusSCl transaction. When such transaction is noticed, a comparator checks if there is an entry in any MSHR matching the received bus transaction. In this case, action is taken so as to resume the pending request.

Due to this feature, it is possible for a cache to have pending requests for a memory location that is not cached or is cached in an invalid state. The location will be brought again into the cache when the synchronization miss is resolved. The ability of replacing cache lines that have pending requests allows efficient management and resuming of pending requests with minimum risk of saturating the cache hierarchy.

A representative scenario is shown in Figure 14, in which three nodes have pending requests to a location (X) in their MSHR. While nodes A and B have invalid copies in their caches, node C has the exclusive ownership of the referred location, whose full/empty state bit is unset. After node C successfully performs a conditional altering write to location X, this event is notified on the bus by a BusSWr transaction. This transaction informs nodes A and B that they can resume the pending request to location X, 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 former will be resumed and the operation at node A will stay pending in the MSHR.

2.4.5 State transition rules in the FE-MESI protocol

A detailed explanation of the new transition rules from each coherence state introduced in the modified MESI cache coherency protocol with integrated fine-grain synchronization is presented in the following sections. A description in the form of C-styled pseudo-code is also presented in each case. Observe that, as with the ordinary coherence misses, the ordering of synchronization misses from different processors is imposed by bus order.

Here we define only a *subset* of state transitions for each initial state of the modified MESI protocol with implicit full/empty states depicted in Figure 12.

1. It could be possible to make read requests be satisfied by pending writes to the same location. However, this introduces extra complexity in the memory unit in order to meet the consistency model. A write request cannot be satisfied by a pending read request in any case.

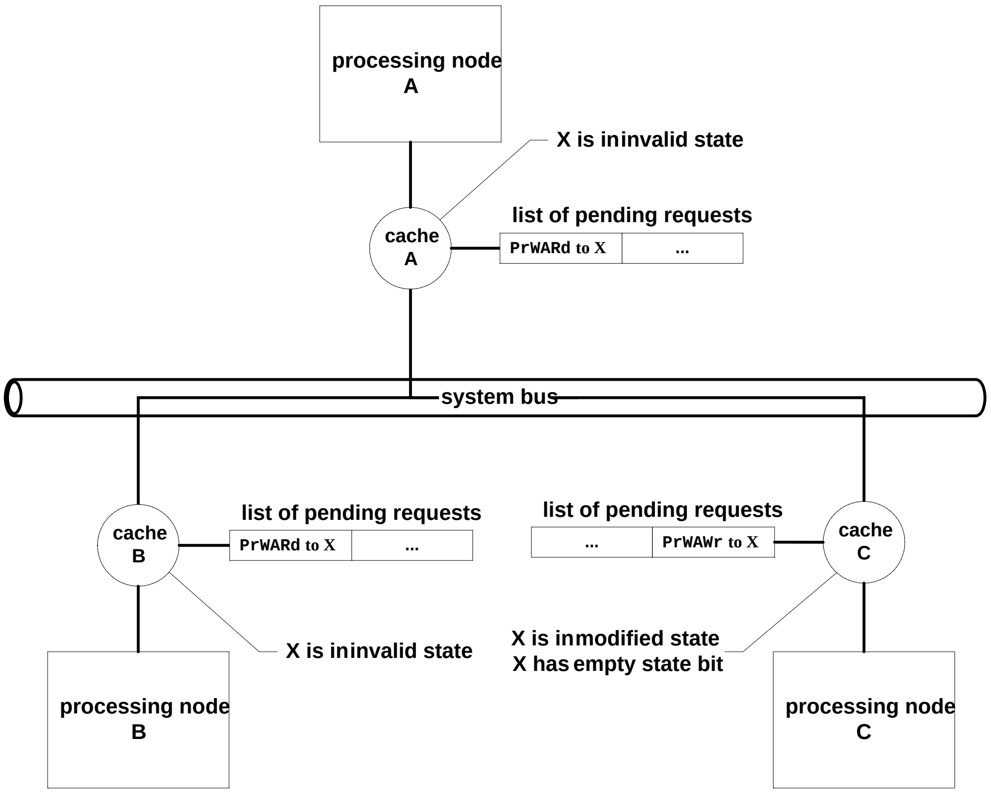


Figure 14: Resuming of pending requests

Invalid State

Assume, that the initial state of both parts of a cache line, i.e. data and FE-state, is *Invalid*.

Figure 15 shows transition rules for the initially invalid cache line. A successful conditional waiting read request from the local processor (PrWNRd) leads either to the *Exclusive-Clean* state (if no other cache holds a copy of the block) or to the *Shared* state (if more caches have a copy of the accessed block). In any case, a BusRd transaction is generated in order to fetch the data from the corresponding cache or shared memory location. However, if the synchronization condition is not met ($\bar{C}$), then the request is appended to the local deferred list and the state is not changed. This occurs when neither the caches nor the shared memory assert the C line.

Cache-to-cache transfers are needed when data is modified in one or more caches and the copy in the shared memory is stale. An alternative is to flush the modified data back to memory and then to the node that requested access, but this approach is obviously slower than the former.

A successful waiting write from the processor (PrWAWr) leads either to the *Modified* state (if no other cache holds a copy of the block) or to the *Shared* state (if more caches have a copy of the block). This implies a performance improvement since the next successful synchronized operation to the same cache line will necessarily be a read and a state transaction will be saved[1]. If the synchronization condition is not met (the line C is asserted), then the operation is suspended.

A PrUACl request generates a BusSCl transaction but does not load the block into cache. This is a design alternative and will be evaluated at the simulation stage of this study.

Modified state

Transition rules for the *Modified* state are shown in Figure 16. If the full/empty bit is set, a conditional waiting read (PrWNRd) retrieves the data from the local cache and generates no bus transaction. Otherwise, the request is appended to the local deferred list. A conditional waiting write (PrWAWr) fails if the C line is asserted and sets the FE-bit otherwise. In the latter, a BusSWr transaction is issued and the relevant pending requests in the local deferred list are resumed. The effect of a BusSWr in the other caches is precisely to set the FE-bit and to analyze their deferred list so as to resume the relevant pending requests.A PrUACl request generates a BusSCl transaction and unsets the FE-bit. This transaction does not flush the block from cache. This is a design alternative and will be evaluated at the simulation stage of this study.

1. If a transition to the *modified* state is performed as in the ordinary MESI protocol, an additional BusRdX transaction is then required on the bus.

```
SWITCH(incomingRequest) {
   CASE PrUNRd: send(BusRd);
                IF (S) {
                   flushFromOtherCache(); nextState = shared;
                } ELSE {
                   readFromMemory(); nextState = exclusive;
                }
                BREAK;
   CASE PrUNWr: send(BusRdX); nextState = modified; BREAK;
   CASE PrWNRd: send(BusRd);
                IF (S && C) {
                   flushFromOtherCache(); nextState = shared;
                } ELSE IF (!S && C) {
                   readFromMemory(); nextState = exclusive;
                } ELSE {
                   addToDeferredList(); // Wait.
                }
                BREAK;
   CASE PrWAWr: send(BusSWr);
                IF (S && !C) {
                   writeToBus(); nextState = shared;  // To be evaluated at simulation.
                } ELSE IF (!S && !C) {
                   writeToCache(); nextState = modified;
                } ELSE {
                   addToDeferredList(); // Wait.
                }
                BREAK;
   CASE PrUACl: IF (C) {
                   send(BusSCl); nextState = invalid;
                }
}
```

Figure 15: Transition rules for the *Invalid* state of a cache line in the FE-MESI protocol

```
SWITCH(incomingRequest) {
   // Processor requests
   CASE PrUNRd: readFromCache(); nextState = modified; BREAK;
   CASE PrUNWr: writeToCache(); nextState = modified; BREAK;
   CASE PrWNRd: IF (full) {
                   readFromCache(); nextState = modified;
                } ELSE {
                   addToDeferredList(); nextState = modified;
                }
                BREAK;
   CASE PrWAWr: send(BusSWr);
                IF (empty) {
                   writeToCache(); resumePendingReqs(); nextState = modified;
                } ELSE {
                   addToDeferredList(); nextState = modified;
                }
                BREAK;
   CASE PrUACl: IF (full) {
                   unsetFE(); nextState = modified;
                }
                BREAK;
   // Bus signals
   CASE BusRd:  flush(); nextState = shared; BREAK;
   CASE BusRdX: flush(); nextState = invalid; BREAK;
   CASE BusSWr: IF (empty) {
                   writeToCache(); resumePendingReqs(); nextState = shared;
                }
                BREAK;
   CASE BusSCl: IF (full) {
                   unsetFE(); nextState = shared;
                }
}
```

Figure 16: Transition rules for the *Modified* state of a cache line in the FE-MESI protocol

Exclusive-clean state

Transition rules for the *Exclusive* state of a cache line are depicted in Figure 17. As no other caches hold a copy of this block, a synchronized read (PrWNRd) leads to the same coherence state.

```
SWITCH(incomingRequest) {
   // Processor requests
   CASE PrUNRd: readFromCache(); nextState = exclusive; BREAK;
   CASE PrUNWr: writeToCache(); nextState = modified; BREAK;
   CASE PrWNRd: IF (full) {
                   readFromCache(); nextState = exclusive;
                } ELSE {
                   addToDeferredList(); nextState = exclusive;
                }
                BREAK;
   CASE PrWAWr: send(BusSWr);
                IF (empty) {
                   writeToCache(); resumePendingReqs();nextState = shared;
                } ELSE {
                   addToDeferredList(); nextState = exclusive;
                }
                BREAK;
   CASE PrUACl: IF (full) {
                   unsetFE(); nextState = modified;
                }
                BREAK;
   // Bus signals
   CASE BusRd:  flush(); nextState = shared; BREAK;
   CASE BusRdX: flush(); nextState = invalid; BREAK;
   CASE BusSWr: IF (empty) {
                   writeToCache(); resumePendingReqs(); nextState = shared;
                }
                BREAK;
   CASE BusSCl: IF (full) {
                   unsetFE(); nextState = shared;
                }
}
```

Figure 17: Transition rules for the *Exclusive* state of a cache line in the FE-MESI protocol

Shared state

If the initial state of a cache line is *Shared* the same rules apply as for the *Modified* state, with the only exception of the BusSWr and BusSCl bus transactions, which do not cause a state transition in this case. Transition rules for the *Shared* state if a cache line are depicted in Figure 18.

2.4.6 Summary

We have developed (in part) a bus based coherence protocol with fine-grain synchronization support in the form of state diagrams as well as pseudo-codes. Although this implementation considers only waiting non-altering reads and waiting altering writes, the behavior of other memory operations is derived in a straightforward manner, as it is a simplified version of the former.

One of the base ideas of the protocol is that full/empty state bit coherence is maintained by bus transactions defined for this purpose, namely BusSWr and BusSCl. An additional bus signal *C* called shared-word is also introduced in order to implement the conditional behavior of synchronizing operations.

A drawback of integrating fine-grain synchronization support at the cache level is the complexity of managing pending synchronization requests. It is expected that this supplementary complexity does not translate in excessive hardware overhead, as most of the required hardware, such as MSHRs or similar, is already present in modern multiprocessors. Consequently, application software making use of synchronizing memory operations will likely experience a noteworthy performance improvement without the need of extensive hardware deployment.

```
SWITCH(incomingRequest) {
   // Processor requests
   CASE PrUNRd: readFromCache(); nextState = shared; BREAK;
   CASE PrUNWr: send(BusRdX); writeToCache(); nextState = modified; BREAK;
   CASE PrWNRd: IF (full) {
                   readFromCache(); nextState = shared;
                } ELSE {
                   addToDeferredList();
                   nextState = shared;
                }
                BREAK;
   CASE PrWAWr: send(BusSWr);
                IF (empty) {
                   writeToCache(); resumePendingReqs();
                   nextState = shared; // To be evaluated at simulation.
                } ELSE {
                   addToDeferredList(); nextState = shared;
                }
                BREAK;
   CASE PrUACl: IF (full) {
                   unsetFE(); send(BusSCl); nextState = shared;
                }
                BREAK;
   // Bus signals
   CASE BusRd:  flush(); nextState = shared; BREAK;
   CASE BusRdX: flush(); nextState = invalid; BREAK;
   CASE BusSWr: IF (empty) {
                   writeToCache(); resumePendingReqs(); nextState = shared;
                }
                BREAK;
   CASE BusSCl: IF (full) {
                   unsetFE(); nextState = shared;
                }
}
```

Figure 18: Transition rules for the Shared state of a cache line in the FE-MESI protocol

2.5 Integration of Fine-Grain Synchronization with A Directory-Based Protocol

In a distributed shared memory multiprocessor, such as the one shown in Figure 19, each shared memory block has a directory entry that lists the nodes that have a cached copy of the data. Full/empty bits are stored as an extra field in the coherence directory entry. Point-to-point messages are used to keep the directory up-to-date and to request permission for a load or a store to a particular location.

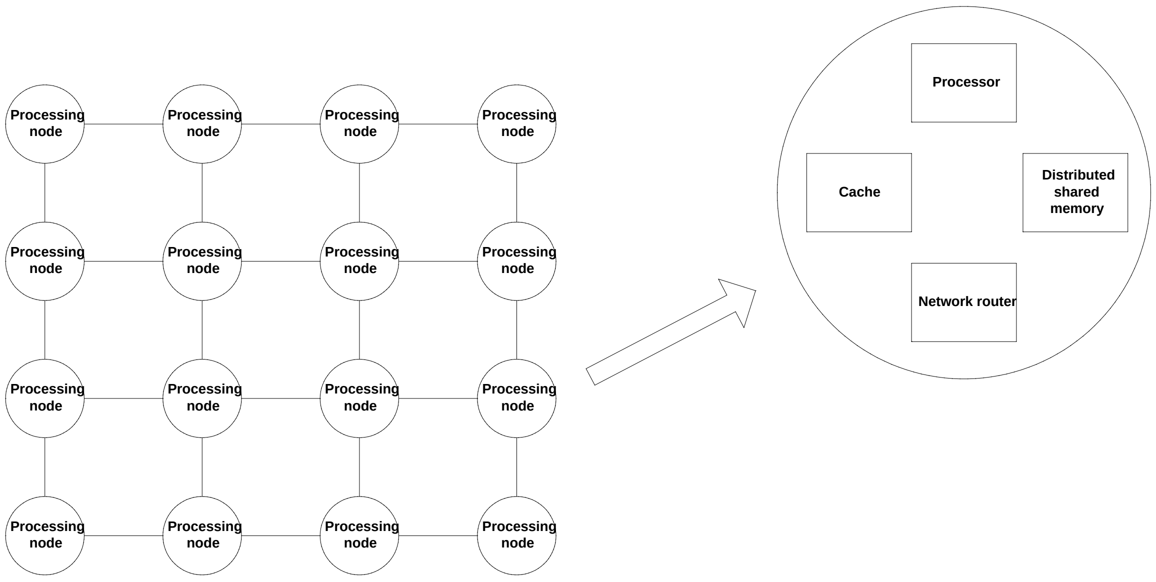


Figure 19: Mesh network-based architecture

The design of a directory-based cache coherency protocol integrated with fine-grained synchronization presented in this report is based on a coherence protocol of the MIT Alewife multiprocessor [8]. We design a limited directory protocol, thus restricting the amount of simultaneous copies of a memory block. The following directory entry states are defined in Alewife's 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 in progress).
- *Write Transaction*: Cache is holding a write request (invalidation in progress).

The state diagram corresponding to this protocol is shown in Figure 20. The semantics of the transitions depicted in this figure are resumed in Table 6 [17].

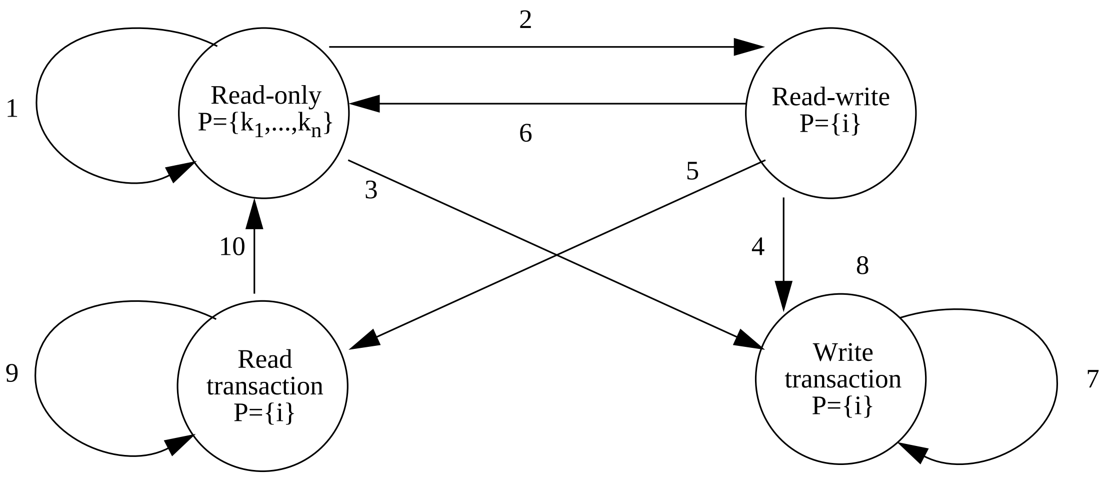


Figure 20: Alewife's coherence protocol state diagram

Table 6: Semantics of the transitions in the directory-based protocol (See Figure 20)

Label	Input message	Output message	Label	Input message	Output message
1	i → RREQ	RDATA → i	8	j → ACKC	WDATA → i
	i → FETCH	RDATA → i		j → REPM	WDATA → i
2	i → WREQ	WDATA → i		j → UPDATE	WDATA → i
	i → MREQ	MODG → i	9	j → RREQ	BUSY → j
3	i → WREQ	INVR → kj		j → WREQ	BUSY → j
	i → MREQ	INVR → kj		j → MREQ	BUSY → j
4	i → WREQ	INVW → i		j → FETCH	BUSY → j
	i → MREQ	INVW → i	10	j → ACKC	RDATA → i
5	j → RREQ	INVW → i		j → REPM	RDATA → i
	j → FETCH	INVW → i		j → UPDATE	RDATA → i
6	i → REPM	-			
7	j → RREQ	BUSY → j			
	j → WREQ	BUSY → j			
	j → MREQ	BUSY → j			
	j → FETCH	BUSY → j			
	j → ACKC	-			

Although the Alewife multiprocessor provides support for fine-grain synchronization, these mechanisms are implemented over the cache coherence protocol, which works as if full/empty bits do not exist. The cache controller in Alewife has limited hardware support for full/empty bits storage. Concretely, these bits are saved as an extra field in the cache tags. This has two advantages. First, the memory used to store cache data does not need to have odd word-length. Second, access to the cache data is slower than access to the cache tags.

When the processor requests a memory access, the Communications and Memory Management Unit (CMMU) determines whether the access is local or remote. The CMMU also checks if the access implies a synchronizing operation by analyzing the ASI value in the memory operation. The address corresponding to the access is checked against the cache tags file, and both the appropriate tag and the full/empty bit are retrieved. At this point one of the following actions is executed[1]:

- a context switch is executed if the access produces a cache miss,
- a full/empty trap is fired in the case of a synchronization fault,
- otherwise, the operation is completed successfully.

According to the performance measures made in Alewife, the overhead of successful synchronizing operations is not significant [15]. When a synchronization miss is detected, a trap is fired and the corresponding thread either polls the location until the synchronization condition is met or blocks according to a given waiting algorithm. While no additional hardware is required for polling, blocking needs to save and restore context registers. The latter case is notably expensive, as loads take two cycles and stores take three cycles.

By integrating synchronization mechanisms with the coherence protocol, the complexity introduced by thread scheduling is avoided. Instead, synchronization misses are handled similarly to ordinary cache misses. As the hardware needed to deal with the latter has already the capability to store part of the information associated with a synchronization miss, it is expected that the hardware overhead introduced by integrating synchronization mechanisms with cache coherence is not excessive.

2.5.1 Mapping between processor instructions and network transactions

The network transactions used in the proposed protocol are explained in Table 7, which shows both messages sent from a cache to memory and requests sent back from memory to a cache.

Six new messages are introduced in order to implement fine-grain synchronization at the cache level. More concretely, 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 the full/empty 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)
	SRDENY	sent if a SRREQ misses; the requesting cache will retry at a later stage
	SWDENY	sent if a SWREQ misses; the requesting cache will retry at a later stage
	INV	invalidates cached words
	ACKSC	acknowledges that the full/empty bit has been unset in all the copies of the block
	BUSY	response to any RREQ or WREQ while invalidations are in progress

1. In all cases, the retrieved full/empty bit is placed into the external condition codes so that the processor has access to it.

As proposed in [22], some fields are needed in the coherence protocol messages in order to integrate fine-grain synchronization. We will make use of some of these proposed additional fields. Specifically, the following fields are required:

- slot's index in the cache line which is being accessed,
- slots in the home directory copy whose list of pending requests is empty; this allows saving protocol messaged in some cases where a block is in the read-write state (see section 0 for more details),
- deferred lists in remote caches are sent to the home node when they release the exclusive ownership; this scenario is further explained in Section 2.5.2.

When a processing node issues a memory operation, the cache located at that node interprets the request and translates it into one or more network transactions. The correspondence between the different processor instructions and memory requests sent over the network is shown in Table 8.

Table 8: Correspondence between processor instructions and memory requests

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

2.5.2 Management of pending requests

Extensive discussion about different alternatives for managing deferred lists is presented in 0. We propose a hybrid procedure for managing deferred lists in which lists of pending operations are kept either at the home directory or in a distributed manner, depending on the state of the line to which pending operations refer. The rules for coalescing requests are the same as in Table 5.

Lists of pending requests for memory locations that are in an *Absent*[1] or *Read-Only* state are maintained as an additional field in the corresponding home directory. Effectively, in these states it is not possible to adopt a distributed approach, since after a transition to the *Read-Write* state the home directory will need to have knowledge of the type of pending requests and the nodes that issued these requests.

A sample case of this scenario is shown in Figure 21, in which two nodes, namely A and B, share a copy of a given memory block. If another node takes the exclusive ownership of this block, information about pending requests issued by nodes A and B will be lost unless the home directory has knowledge of those requests. A naive approach is to make the directory keep track of only the nodes with pending requests, because this would require informing all of these nodes each time a full/empty state change is detected, thus generating extra traffic. Figure 21 also shows a different memory block for which there is no copy at any other node in the system, thus being in the absent state. The same rules apply for this location.

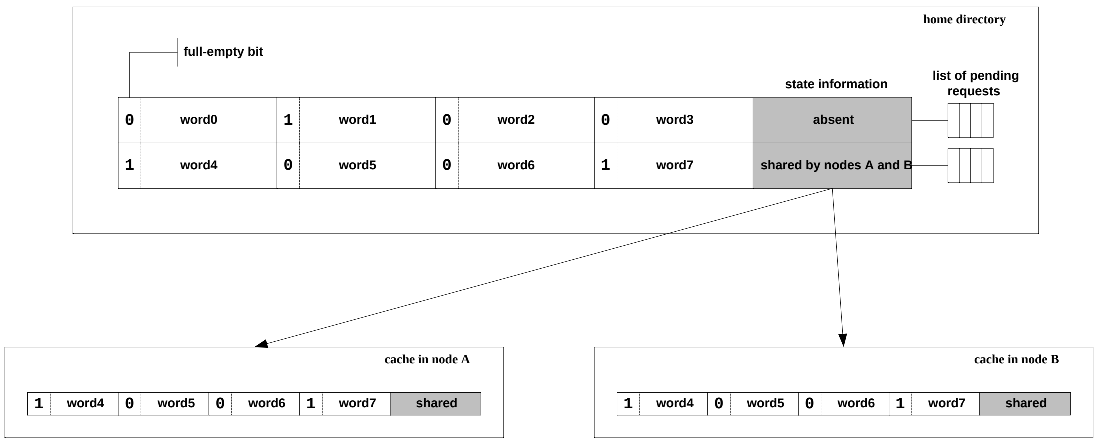
For locations in a *Read-Write* state, we adopt a distributed solution in which both the home directory and the remote cache keep track of pending operations. When a remote cache releases its copy of the block, the deferred list kept locally to that cache is sent to the home node and merged with the deferred list at the home directory. The rules for coalescing requests are those in Table 5. An example in which a location is first owned by node A and then flushed from its cache is shown on Figure 22.

As in the bus-based scheme, it is also necessary to specify how resuming of pending requests is done. Contrary to the former, coherence of full/empty state bits is not always ensured at the home directory. In fact, the home directory does not have a valid copy of the full/empty bit of a memory location that is in the *Read-Write* state. In such case, the directory forwards requests from other nodes to the exclusive owner of the block, where they will be serviced. According to these features, resuming of pending requests is based on the following rules:

1. The *Absent* state indicates that no cache is holding a copy of the referred memory location. Consequently this location does not fall into any of the four states described on page 23.

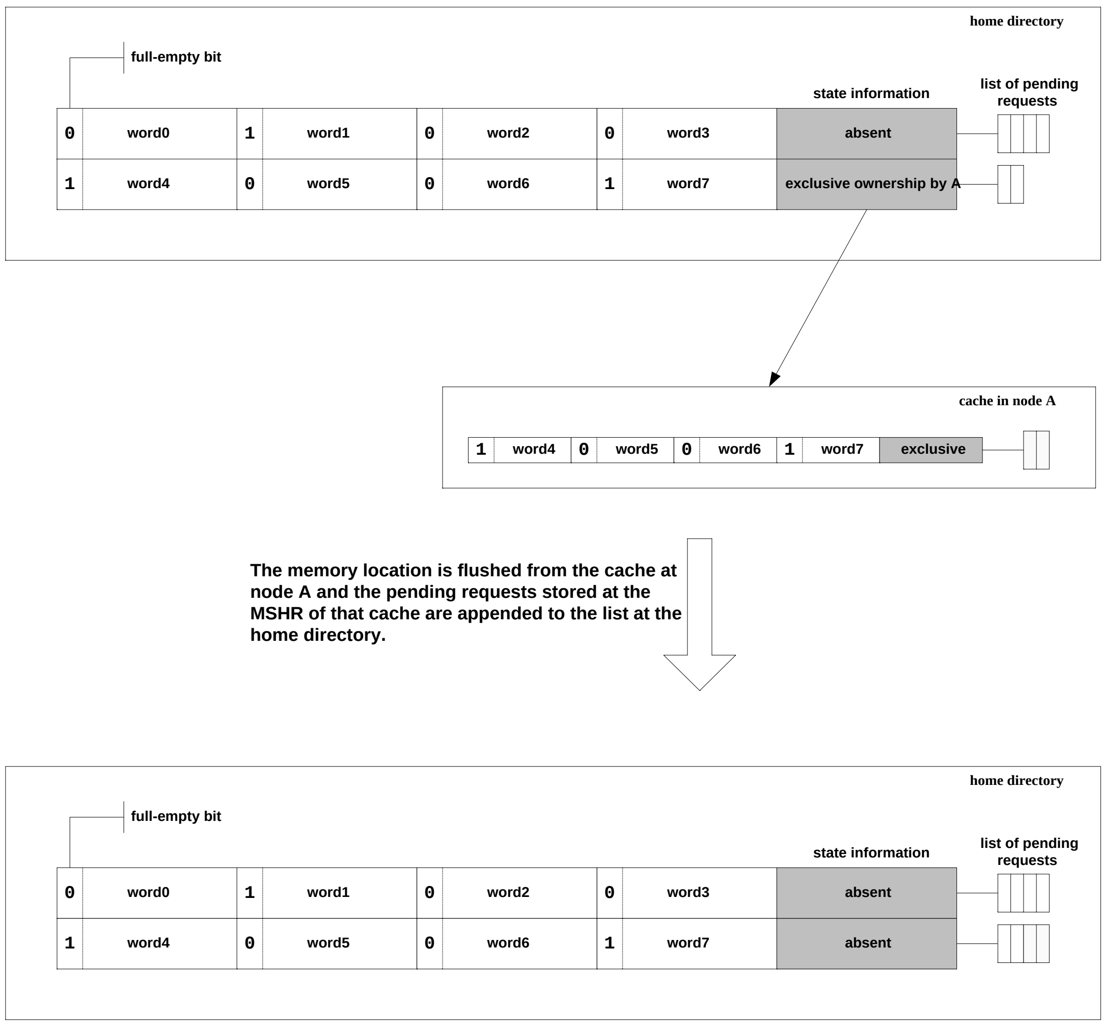
- if a block is in the *Absent* or *Read-Only* state, the home directory is responsible for resuming requests, by checking if there is any entry in the deferred list that matches an incoming transaction,
- if a block is in the *Read-Write* state, the cache having the exclusive ownership knows whether there are pending requests for that block at the home directory. In that case, relevant operations performed at that node are forwarded to the home node in order to check if any pending request can be resumed. Otherwise, the deferred list can be locally managed at the exclusive owner.

Consequently, it is not possible for a cache to have pending requests for a memory location that is not cached. These pending requests are kept and managed at the home directory. This solution is a hybrid approach between a fully distributed and a centralized deferred list management.



home directory
full-empty bit
state information
list of pending requests
0 word0 1 word1 0 word2 0 word3 absent
1 word4 0 word5 0 word6 1 word7 shared by nodes A and B
cache in node A
1 word4 0 word5 0 word6 1 word7 shared
cache in node B
1 word4 0 word5 0 word6 1 word7 shared

Figure 21: Management of pending requests for an absent or read-only memory block



home directory
full-empty bit
state information
list of pending requests
0 word0 1 word1 0 word2 0 word3 absent
1 word4 0 word5 0 word6 1 word7 exclusive ownership by A
cache in node A
1 word4 0 word5 0 word6 1 word7 exclusive
The memory location is flushed from the cache at node A and the pending requests stored at the MSHR of that cache are appended to the list at the home directory.
home directory
full-empty bit
state information
list of pending requests
0 word0 1 word1 0 word2 0 word3 absent
1 word4 0 word5 0 word6 1 word7 absent

Figure 22: Management of pending requests for a read-write memory block

2.5.3 Directory transition rules

A detailed explanation of the transition rules from each coherence state is presented in the following sections. The rules are also presented in the form of C-styled pseudo-code. In the state diagrams, the following notation is used.

```
processor id: input message, preconditions / side effects / output message, local actions
```

A tilde symbol (~) indicates no side effect or output message.

Absent State

Transition rules for the *Absent* state of a directory entry are depicted in Figure 23. State transitions from the *Absent* state are depicted in Figure 24.

If a SRREQ[1] is received and the synchronization state is met (the FE bit is set), the requesting cache is added to the directory and the requested data is sent in a RDATA message. The state is then changed to *Read-Only*. If the SRREQ fails (the FE bit is unset), a RDENY message is sent back to the requesting cache and the operation is appended to the deferred list in the home node. The state is not changed and the requesting cache waits until the home node solves the synchronization miss and sends back the requested data.

If a SWREQ is received and the synchronization condition is met (FE is not set), the requesting cache is added to the directory and a WDATA message is sent to it. Any relevant pending request in the local deferred list is resumed and appropriate data is sent to its corresponding cache, which is also added to the directory. The state is then changed to *Read-Only*, and not to *Read-Write* as it could be expected. This optimization allows other processing nodes to read this data without any state transition. If the full/empty bit is set on a SWREQ, then a WDENY is replied and the operation is suspended. The state in the home directory is not changed.

If a SCREQ is received, the FE-bit is reset and an ACKSC sent back to the requesting cache. The state in the home directory is not changed.

Read-Only State

Transition rules for the *Read-Only* state of a directory entry are depicted in Figure 25. State transitions from the *Read-Only* state are depicted in Figure 26.

If a SRREQ is received and the synchronization condition is met, an RDENY message is replied and the request is appended to the local deferred list. If the synchronization condition is met and the requesting cache is already in the directory, an RDATA message is sent back with the requested memory location. If the requesting cache is not in the directory and there are still free directory entries, the cache is added to the directory. Otherwise, a random cache is replaced with the requesting cache and an INV message is sent to the removed cache. The home directory state is not changed in any case.

If a SWREQ is received and the full/empty is set, then a WDENY message is replied. Otherwise, the requesting cache is added to the directory and a WDATA message is sent back. In any case, the home directory state is not changed.

If a SCREQ is received and no more caches share this block, then the full/empty bit is cleared and the request is acknowledged with an ACKSC message. The state is not changed in this case. However, if more caches have a copy of this block, their full/empty bits must be reset before acknowledging the operation. Consequently, the state is changed to *Write-Transaction* and an SCREQ message is sent to each cache with a copy of the block. Note that the SCREQ operation is particularly time-expensive, as it works as a barrier for all the involved caches.

1. See network transactions in Table 7 on page 24.

```
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();
                     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); resumePendingReqs();
                     nextState = readOnly;
                  } ELSE {
                     send(WDENY, i); addToDeferredList();
                     nextState = absent;
                  }
                  BREAK;
   CASE SCREQ(i): unsetFE(); send(ACKSC, i);
                  nextState = absent;
                  BREAK;
   CASE ACKC(i):  ackCounter--;
                  nextState = absent;
}
```

Figure 23: Transition rules for the *Absent* state of a directory entry in the FE-limited-directory protocol.

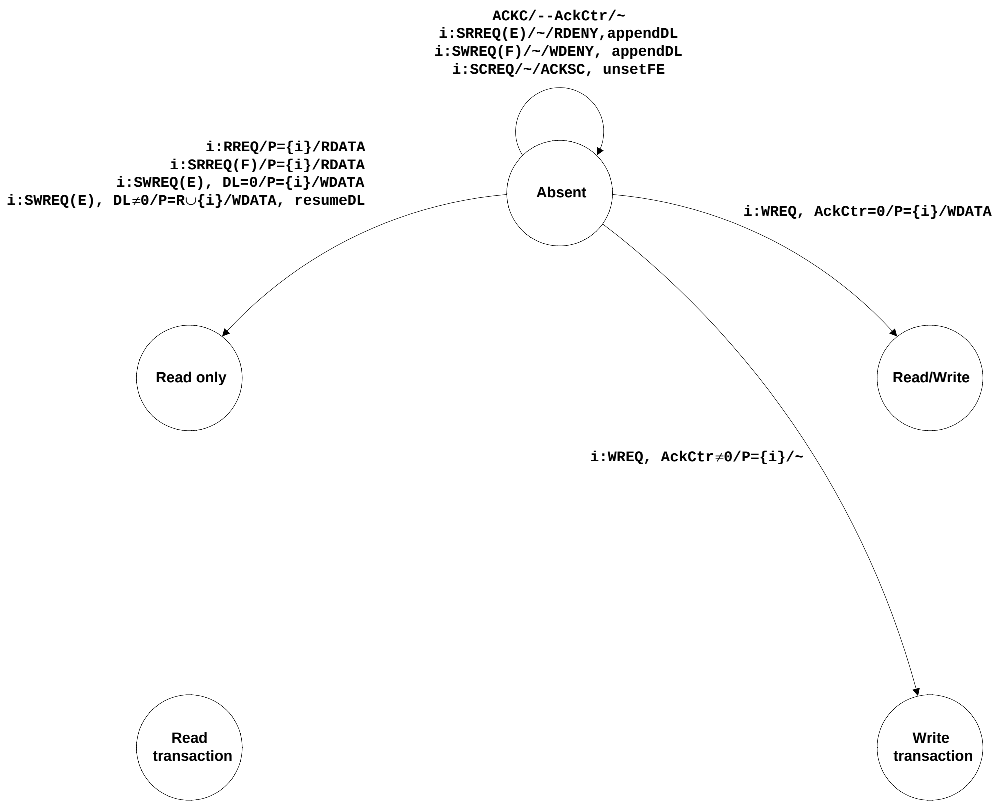


Figure 24: State transitions from the *Absent* state of a directory entry in the FE-limited-directory protocol

```
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(); // j is the evicted line.
                     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 IF (empty) {
                     send(RDENY, i);
                     addToDeferredList();
                  }
                  nextState = readOnly;
                  BREAK;
   CASE SWREQ(i): IF (empty & deferredListEmpty()) {
                     addNodeToDirectory(i);
                     send(WDATA, i);
                  } ELSE IF (empty & !deferredListEmpty()) {
                     addNodeToDirectory(i);
                     send(WDATA, i);
                     resumePendingReqs();
                  } ELSE {
                     send(WDENY, i);
                     addToDeferredList();
                  }
                  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)) {
                     unsetFE();
                     send(ACKSC, i);
                     nextState = readOnly;
                  }
                  BREAK;
   CASE ACKC(i):  ackCounter--;
                  nextState = readOnly;
}
```

Figure 25: Transition rules for the *Read-Only* state of a directory entry in the FE-limited-directory protocol

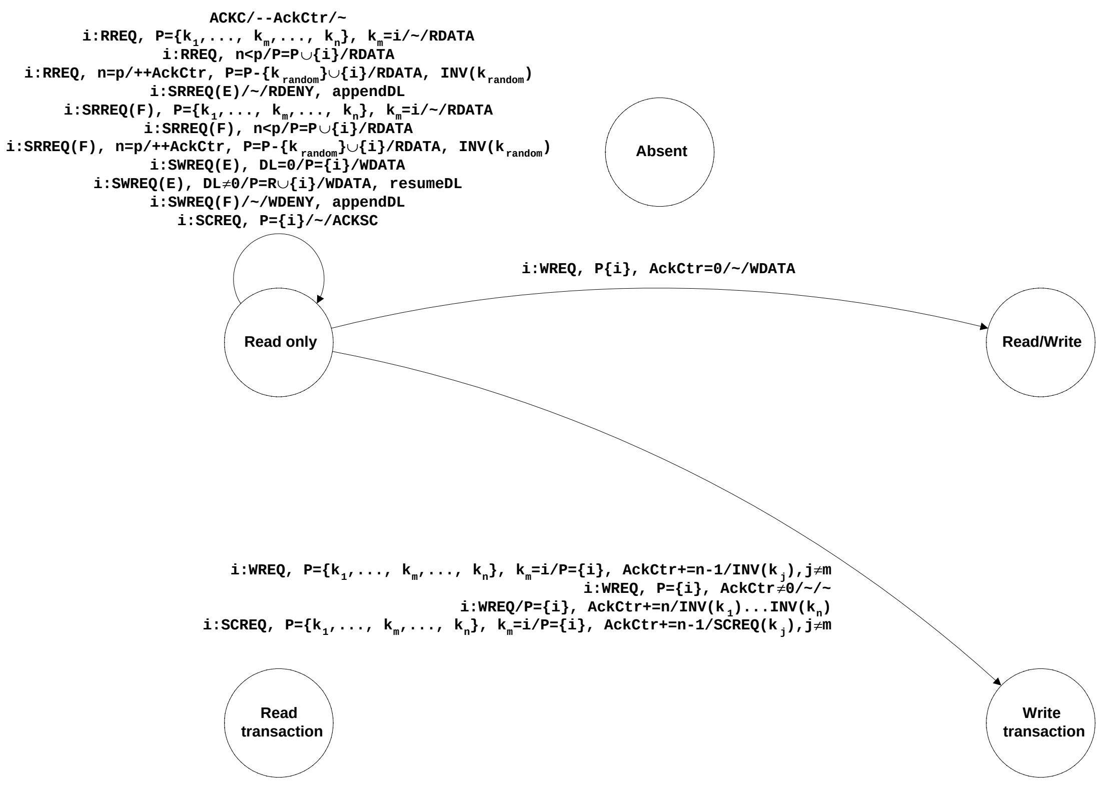

ACKC/--AckCtr/~
i:RREQ, P={k_1,..., k_m,..., k_n}, k_m=i/~/RDATA
i:RREQ, n<p/P=P∪{i}/RDATA
i:RREQ, n=p/++AckCtr, P=P-{k_{random}}∪{i}/RDATA, INV(k_{random})
i:SRREQ(E)/~/RDENY, appendDL
i:SRREQ(F), P={k_1,..., k_m,..., k_n}, k_m=i/~/RDATA
i:SRREQ(F), n<p/P=P∪{i}/RDATA
i:SRREQ(F), n=p/++AckCtr, P=P-{k_{random}}∪{i}/RDATA, INV(k_{random})
i:SWREQ(E), DL=0/P={i}/WDATA
i:SWREQ(E), DL≠0/P=R∪{i}/WDATA, resumeDL
i:SWREQ(F)/~/WDENY, appendDL
i:SCREQ, P={i}/~/ACKSC
Absent
i:WREQ, P{i}, AckCtr=0/~/WDATA
Read only
Read/Write
i:WREQ, P={k_1,..., k_m,..., k_n}, k_m=i/P={i}, AckCtr+=n-1/INV(k_j),j≠m
i:WREQ, P={i}, AckCtr≠0/~/~
i:WREQ/P={i}, AckCtr+=n/INV(k_1)...INV(k_n)
i:SCREQ, P={k_1,..., k_m,..., k_n}, k_m=i/P={i}, AckCtr+=n-1/SCREQ(k_j),j≠m
Read transaction
Write transaction

Figure 26: State transitions from the *Read-Only* state of a directory entry in the FE-limited-directory protocol

Read-Write State

Transition rules for the *Read-Write* state of a directory entry are depicted in Figure 27. State transitions from the *Read-Write* state are depicted in Figure 28.

If a SRREQ is received, either a RDATA or RDENY message is replied depending on whether the synchronization condition is met. The state is not changed in any case.

If a SWREQ is received from a cache other than the owner of the block and the synchronization condition is met, the request is forwarded to the owner. Additional functionality is required in the cache protocol, as the owner is expected to answer this type of forwarded requests. Another design alternative is to centralize all the synchronized writes at the home node. This avoids the need of forwarded requests but introduces an overhead associated with the excess traffic generated by the caches that request a synchronized write to the home node even though they are exclusive owners for that block.

Our design assumes that caches can service forwarded requests. In this case, when a cache with exclusive ownership performs a synchronized write to a block, it notifies the home node only when the deferred list at the home node is not empty. This knowledge is implicit in an extra bit at the cache side, which is set when there are pending requests for the referred location at the home directory, as proposed in [22].

As with the SWREQ operation, a SCREQ coming from a cache different than the owner is forwarded to the cache that has the read-write privilege.

Read-Transaction State

Transition rules for the *Read-Transaction* state of a directory entry are depicted in Figure 29. State transitions from the *Read-Transaction* state are depicted in Figure 30. No new transitions are added from this state. All synchronized operations are ignored and a BUSY message is sent back to the requesting cache.

Write-Transaction state

Transition rules for the *Write-Transaction* state of a directory entry are depicted in Figure 31. State transitions from the *Write-Transaction* state are depicted in Figure 32. As in the previous case, all SRREQ, SWREQ and SCREQ messages are replied with a BUSY message. A new transition is specified in the protocol for the case when a cache has requested to clean the full/empty bit of all caches with a copy of a block. In this case the acknowledgment counter is used to keep track of the caches that have not yet cleared their copy of the FE-bit. After a cache clears its copy of the FE-bit, it sends an ACKSC message to the directory and invalidates the corresponding cache line. When all the ACKSC messages have been received from the home node, then the operation is acknowledged to the requesting cache.

```
SWITCH (incomingRequest) {
   CASE RREQ(j):  IF (!hasPointerInDirectory(j)) { // there is only one node
                     ++ackCounter;                 // in the directory (the
                     send(INV, i);                 // owner, namely "i")
                     clearDirectory(); addNodeToDirectory(j);
                     nextState = readTransaction;
                  }
                  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();
                     nextState = readWrite;
                  }
                  BREAK;
   CASE SWREQ(j): IF (!hasPointerInDirectory(j) && empty) {
                     ++ackCounter;
                     send(INV, i); clearDirectory(); addNodeToDirectory(j);
                     nextState = writeTransaction;
                  } ELSE IF (full) {
                     send(WDENY, j); addToDeferredList();
                     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); resumePendingReqs();
                          nextState = readOnly;
}
```

Figure 27: Transition rules for the *Read-Write* state of a directory entry in the FE-limited-directory protocol

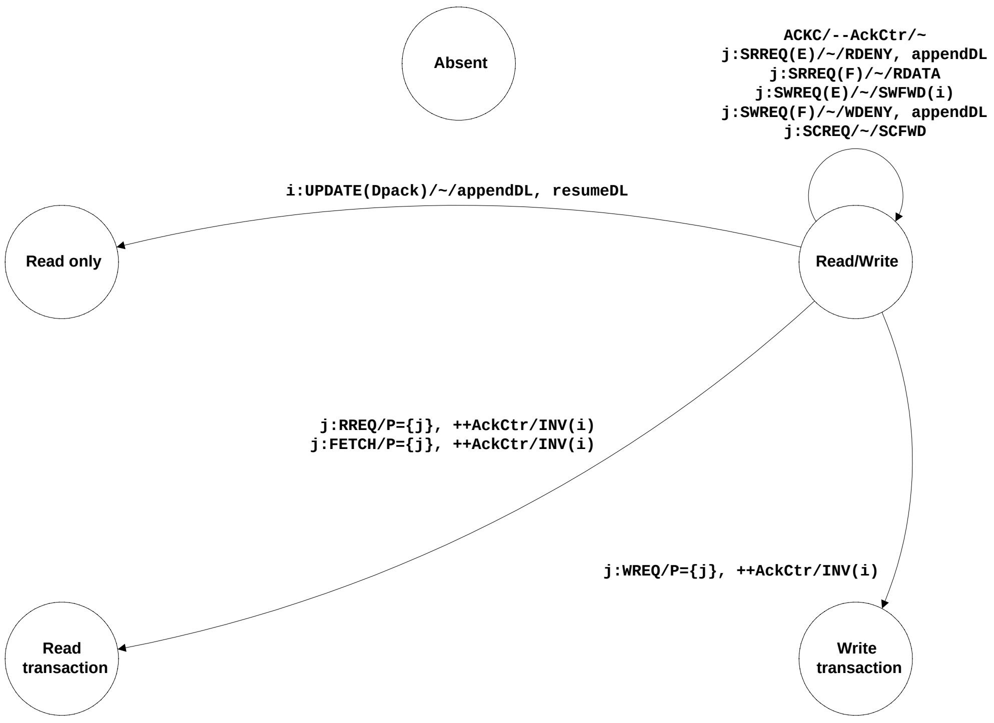

Absent
ACKC/--AckCtr/~
j:SRREQ(E)/~/RDENY, appendDL
j:SRREQ(F)/~/RDATA
j:SWREQ(E)/~/SWFWD(i)
j:SWREQ(F)/~/WDENY, appendDL
j:SCREQ/~/SCFWD
i:UPDATE(Dpack)/~/appendDL, resumeDL
Read only
Read/Write
j:RREQ/P={j}, ++AckCtr/INV(i)
j:FETCH/P={j}, ++AckCtr/INV(i)
j:WREQ/P={j}, ++AckCtr/INV(i)
Read transaction
Write transaction

Figure 28: State transitions from the *Read-Write* state of a directory entry in the FE-limited-directory protocol

```
SWITCH (incomingRequest) {
   CASE RREQ(i):
   CASE WREQ(i):
   CASE SRREQ(i):
   CASE SWREQ(i):
   CASE SCREQ(i):  send(BUSY, i);
                   nextState = readTransaction;
                   BREAK;
   CASE ACKC(i):   ackCounter--;
                   nextState = readOnly;
                   BREAK;
   CASE UPDATE(i): --ackCounter;
                   send(RDATA, i);
                   nextState = readOnly;
}
```

Figure 29: Transition rules for the *Read-Transaction* state of a directory entry in the FE-limited-directory protocol

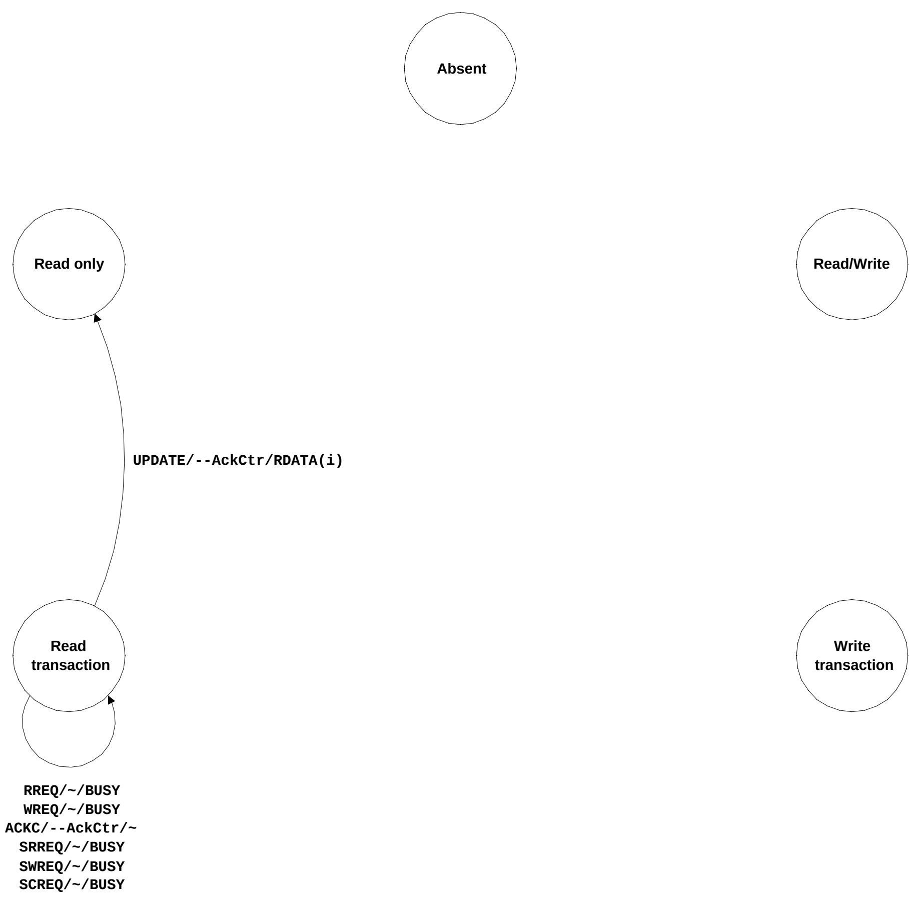
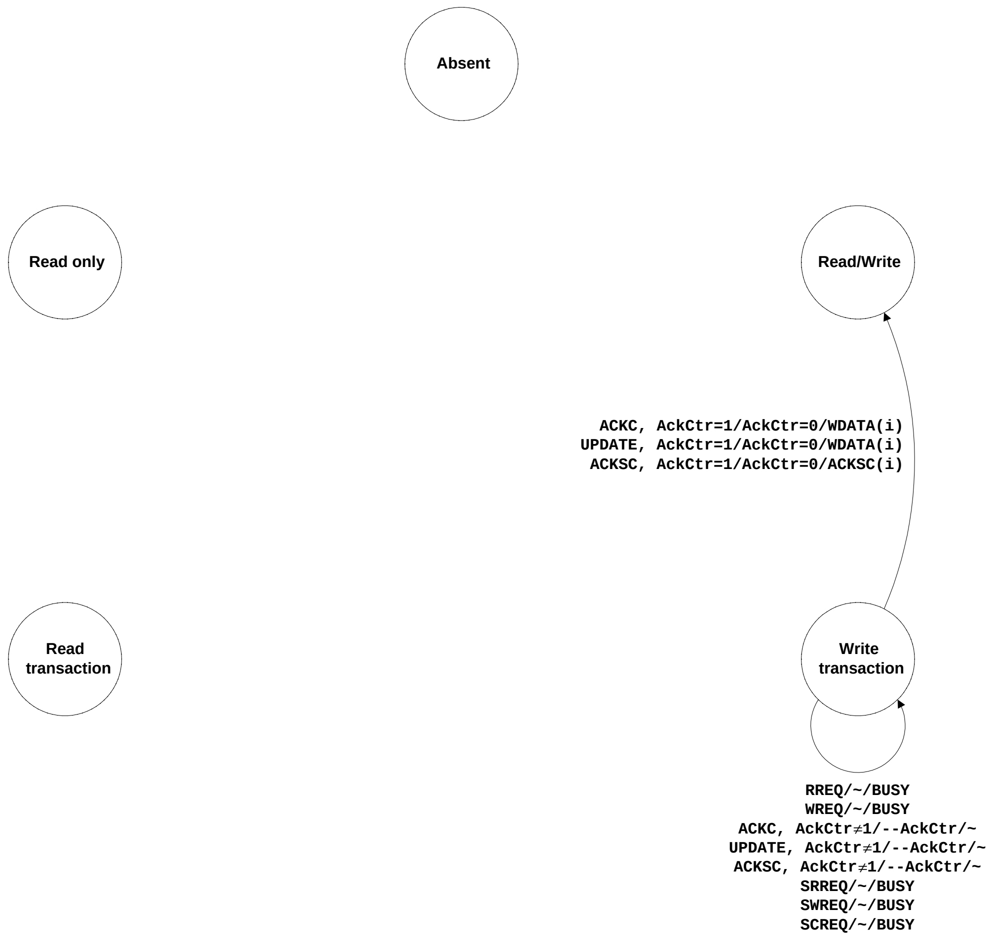


Figure 30: State transitions from the *Read-Transaction* state of a directory entry in the FE-limited-directory protocol

```
SWITCH (incomingRequest) {
   CASE RREQ(i):
   CASE WREQ(i):
   CASE SRREQ(i):
   CASE SWREQ(i):
   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;
                   }
}
```

Figure 31: Transition rules for the *Write-Transaction* state of a directory entry in the FE-limited-directory protocol



Absent
Read only
Read/Write
ACKC, AckCtr=1/AckCtr=0/WDATA(i)
UPDATE, AckCtr=1/AckCtr=0/WDATA(i)
ACKSC, AckCtr=1/AckCtr=0/ACKSC(i)
Read transaction
Write transaction
RREQ/~/BUSY
WREQ/~/BUSY
ACKC, AckCtr≠1/--AckCtr/~
UPDATE, AckCtr≠1/--AckCtr/~
ACKSC, AckCtr≠1/--AckCtr/~
SRREQ/~/BUSY
SWREQ/~/BUSY
SCREQ/~/BUSY

Figure 32: State transitions from the *Write-Transaction* state of a directory entry in the FE-limited-directory protocol

2.5.4 Summary

A directory based protocol with support for fine-grain synchronization has been systematically specified in the form of state diagrams and pseudo-code. As in the bus based directory protocol (see Section 2.4), only waiting non-altering reads and waiting altering writes are considered in this implementation. The operation of other variants of synchronized accesses can be easily inferred because they are a simplified version of the former.

Six new network messages are introduced in order to implement fine-grain synchronization at the cache coherence level. Some optimizations reducing the number of messages are proposed, requiring additional functionality in the protocol so that caches can service forwarded requests that are sent to the directory from other caches.

We propose a deferred list management scheme in which lists of pending requests can be either kept at the home directory or distributed between the directory and the caches. This solution is a compromise between a distributed approach and a centralized design and minimizes the number of protocol messages sent over the network. The same rules as in the bus-based approach are applied for coalescing of pending requests.

3 Simulation

As a practical working model of the proposed coherence protocols, a directory-based protocol with fine-grain synchronization support has been partially implemented and simulated. This experimental model is based on the Rice Simulator for ILP Multiprocessors (RSIM[1]) simulator and runs on Solaris 2.5 or above.

3.1 Features of the simulated platform

RSIM is a discrete event-driven simulator based on the YACSIM library [14]. This means that most of the resources in the simulated architecture are activated as events only when they have some tasks to execute. As an exception, both processor and caches are simulated as an event that is executed on every cycle. This decision is based on the facts that those units are likely to have nearly continuous activity.

Figure 33 shows the network and memory system hierarchy in the simulation platform.

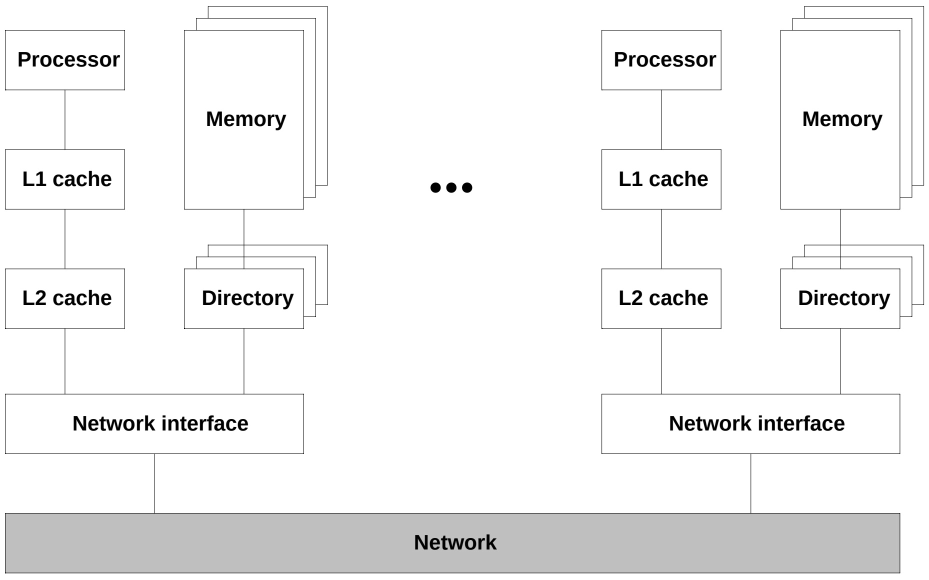


Figure 33: Simulated system architecture

The key features of simulated systems are listed below [19].

- Multiple instruction issue and out-of-order scheduling;
- Branch prediction support;
- Non-blocking loads and stores;
- Optimized memory consistency implementations;
- Two-level cache hierarchy and multiple outstanding cache requests;

1. Available at http://rsim.cs.uiuc.edu/rsim/ (accessed November 2001).

- CC-NUMA shared-memory system;
- Directory-based cache coherence protocol with fine-grain synchronization support;
- Routed two-dimensional mesh network.

Additionally, contention effects are modeled at all resources in the processor, caches, memory banks, processor-memory bus and network.

3.2 Preparing binary code for simulation

The steps required to perform a general simulation with the developed platform are depicted in Figure 34.

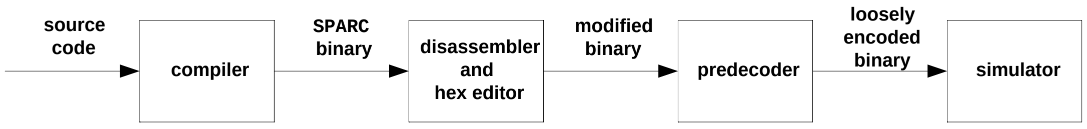


Figure 34: Simulation steps

The starting point is the source code of the program to be run under the simulator. As this code is supposed to successfully compile on an ordinary compiler, no language-level support for expressing data-level synchronization operations is available at this step. However, it is necessary to somehow distinguish these operations. Using unique assembler instructions thorough the source code achieve this. For example, with calls to the function `asm` in the C language.

An 8-bit Alternate Space Indicator (ASI) is defined in the SPARC architecture in order to tag loads and stores with 256 different values. As some of these values are user-defined, they can be used for synchronizing instructions. As a consequence, synchronizing operations are distinguished by particular ASI values. The ASI parameter determines the specific variant of synchronizing instruction that will be executed. A sample C program is presented below.

```
int main(int argc, char **argv) {
  int sVar; /* synchronized variable */
  /* The values of ASI for synchronizing operations are:
     ASI_UE   0x91
     ASI_FF   0x96
     ASI_EF_T 0x9d
  */
  asm("wr   %g0, 0x9d, %asi");     /* WRASI instruction */
  sVar = 5;                        /* synchronized store (STWA_EFT) */

  /* The complete assembler sequence looks like this:
     wr   %g0, 0x9d, %asi
     mov  xxx, %o0                !xxx is the data to be stored
     stwa %o0, [%fp - yyy], %asi  !yyy is an appropriate offset
  */
}
```

In order to get a binary SUN’s `cc` compiler has to be used with the `-xarch=v8plusa` option. Otherwise WRASI will not be recognized as a valid instruction. A binary is obtained with:

```
cc -xarch=v8plusa synch.c -o synch
```

Currently, we modify the op-code of the desired memory instructions (in order to make them synchronized) “manually” using a hexadecimal editor (refer to Appendix A). As stated above, these memory instructions are easily recognized because they are preceded by a write to the ASI register. A disassembler must be used in conjunction with the hexadecimal editor in order to determine the appropriate offset of the memory operations in the binary file. As no SPARC32+ disassembler is openly available, a standard SPARC disassembler was modified in order to recognize the new instructions. A future improvement would be to extend the compiler in order to support the complete set of synchronizing memory instructions. With this extension, the simulation steps would be simplified as shown in Figure 35.

Once a binary with synchronizing instructions is obtained, a predecoder is executed on it. A new binary in a loosely encoded format, which can be interpreted by the simulator, is thus obtained.

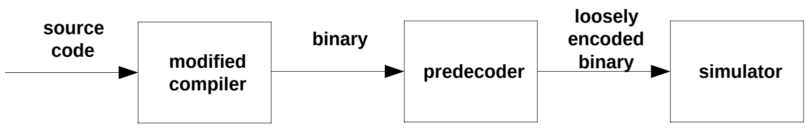


Figure 35: Simulation steps with a compiler supporting synchronizing instructions

3.3 Implementation of synchronizing instructions

As specified in [21], the only SPARC instructions that access memory are load, store, prefetch, swap, and compare-and-swap. An implicit ASI value is provided by normal load and store. On the contrary, an explicit ASI is provided by alternate load and store. This explicit value is given either in the ASI register or in the `imm_asi` instruction field (see Figure 36). The 6-bit field `op3` determines the specific load or store instruction.

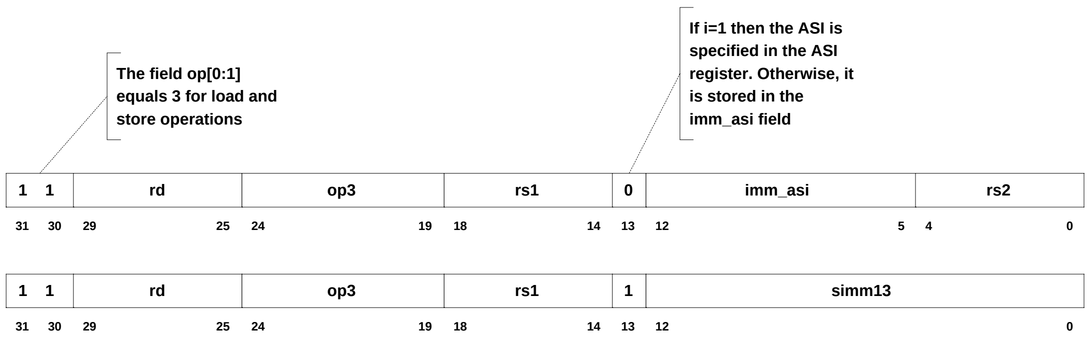

The field op[0:1] equals 3 for load and store operations
If i=1 then the ASI is specified in the ASI register. Otherwise, it is stored in the imm_asi field
1 1 rd op3 rs1 0 imm_asi rs2
31 30 29 25 24 19 18 14 13 12 5 4 0
1 1 rd op3 rs1 1 simm13
31 30 29 25 24 19 18 14 13 12 0

Figure 36: Alternate load and store instruction format

Synchronization instructions are implemented on SPARC by defining them as colored load and stores, as specified by the ASI field. ASI values corresponding to synchronizing instructions are presented in Table 9. Each instruction category is assigned four consecutive ASI values. Two of these values specify altering instructions, while the other two represent non-altering accesses. A total of 16 synchronizing memory instructions are introduced. Although loads and stores are defined on both integers and floating-point data supporting byte, half-word (16-bit), word (32-bit), double-word (64-bit) and quad-word (128-bit) accesses, only integer word memory operations are supported by the simulation platform.

Table 9: ASI values for synchronizing operations

ASI value range	Instruction category
0x90 to 0x93	Unconditional
0x94 to 0x97	Conditional waiting
0x98 to 0x9B	Conditional non-faulting
0x9C to 0x9F	Conditional faulting

Table 10 summarizes the most relevant modifications made to RSIM.

Table 10: Specific modifications made to RSIM

Source file	Changes
`MemSys/cache2.c`	Added extra fields in cache lines storing full/empty bits[a]
`MemSys/directory.c`	Added extra fields in directory storing full/empty bits
`MemSys/l1cache.c`	Cache behavior integrated with full/empty bits and synchronizing operations
`MemSys/mshr.c`	Specification of new types of memory operations and extension of MSHR registers[b]
`MemSys/setup_cohe.c`	Implementation of a coherence protocol integrated with fine-grain synchronization at both L1 and L2 caches
`Processor/except.cc`	New type of soft exception fired my trapping conditional instructions
`Processor/funcs.cc`	Functional implementation of conditional instructions
`Processor/memunit.cc`	Behavior of the memory unit when dealing with synchronizing instructions
`Processor/units.cc`	Specification of functional units used by synchronizing instructions, access types and latencies
`predecode/predecode_instr.cc`	New values of ASI for synchronizing instructions
`predecode/predecode_table.cc`	Association between new instructions and the functions corresponding to its implementation

a. Currently, the simulation platform only supports cache lines of 64-bit length.
b. The management of synchronizing pending requests has not been implemented yet.

Table 11 depicts the set of full/empty memory instructions.

Table 11: Set of Full/Empty Memory Instructions[a]

Alternate superspace			Alternate space	ASI	Instruction	Operation (atomic)	RSIM (extension)
ASI_UNCOND (0x90..0x93)			ASI_UU	0x90	unconditional load or store	set F/E to F/E condition bit; read or write;	LD*_UU ST*_UU
			ASI_UF	0x93	unconditional store and set full	set F/E to F/E condition bit; write and set full;	ST*_UF
			ASI_UE	0x91	unconditional load and set empty	set F/E to F/E condition bit; read and set empty;	LD*_UE
			ASI_UR	0x92	reserved	-	-
ASI_COND (0x94..0x9F)	ASI_COND_WAIT (0x94..0x97)		ASI_EE	0x94	waiting conditional store from empty	write when empty;	ST*_EE
			ASI_EF	0x95	waiting conditional altering store from empty	(write and set full) when empty;	ST*_EF
			ASI_FE	0x96	waiting conditional altering load from full	(read and set empty) when full;	LD*_FE
			ASI_FF	0x97	waiting conditional load from full	read when full;	LD*_FF
	ASI_COND_NOWAIT (0x98..0x9F)	ASI_COND_NOFAULT (0x98..0x9B)	ASI_EE_N	0x98	non-faulting conditional store from empty	set F/E to F/E condition bit; write if empty else skip;	ST*_EEN
			ASI_EF_N	0x99	non-faulting conditional altering store from empty	set F/E to F/E condition bit; (write and set full) if empty else skip;	ST*_EFN
			ASI_FE_N	0x9A	non-faulting conditional altering load from full	set F/E to F/E condition bit; (read and set empty) if full else skip;	LD*_FEN
			ASI_FF_N	0x9B	non-faulting conditional load from full	set F/E to F/E condition bit; read if full else skip;	LD*_FFN
		ASI_COND_FAULT (0x9C..0x9F)	ASI_EE_T	0x9C	faulting conditional store from empty	set F/E to F/E condition bit; write if empty else trap;	ST*_EET
			ASI_EF_T	0x9D	faulting conditional altering store from empty	set F/E to F/E condition bit; (write and set full) if empty else trap;	ST*_EFT
			ASI_FE_T	0x9E	faulting conditional altering load from full	set F/E to F/E condition bit; (write and set full) if empty else trap;	LD*_FET
			ASI_FF_T	0x9F	faulting conditional load from full	set F/E to F/E condition bit; read if full else trap;	LD*_FFT

a. An asterisk indicates a data type, such as floating point (F) or unsigned word (UW). Note that all instructions correspond to the set of LD*A operations in SPARC.

3.4 Simulation flowchart

Figure 37 shows the different stages of a single simulation. The names of the functions and the source code file names corresponding to the listed operations are specified at the top of the boxes. The core of the simulation platform consists of a loop in which scheduled events are executed. The iterations through this loop continue until the event list is empty, meaning that the simulation has finished.

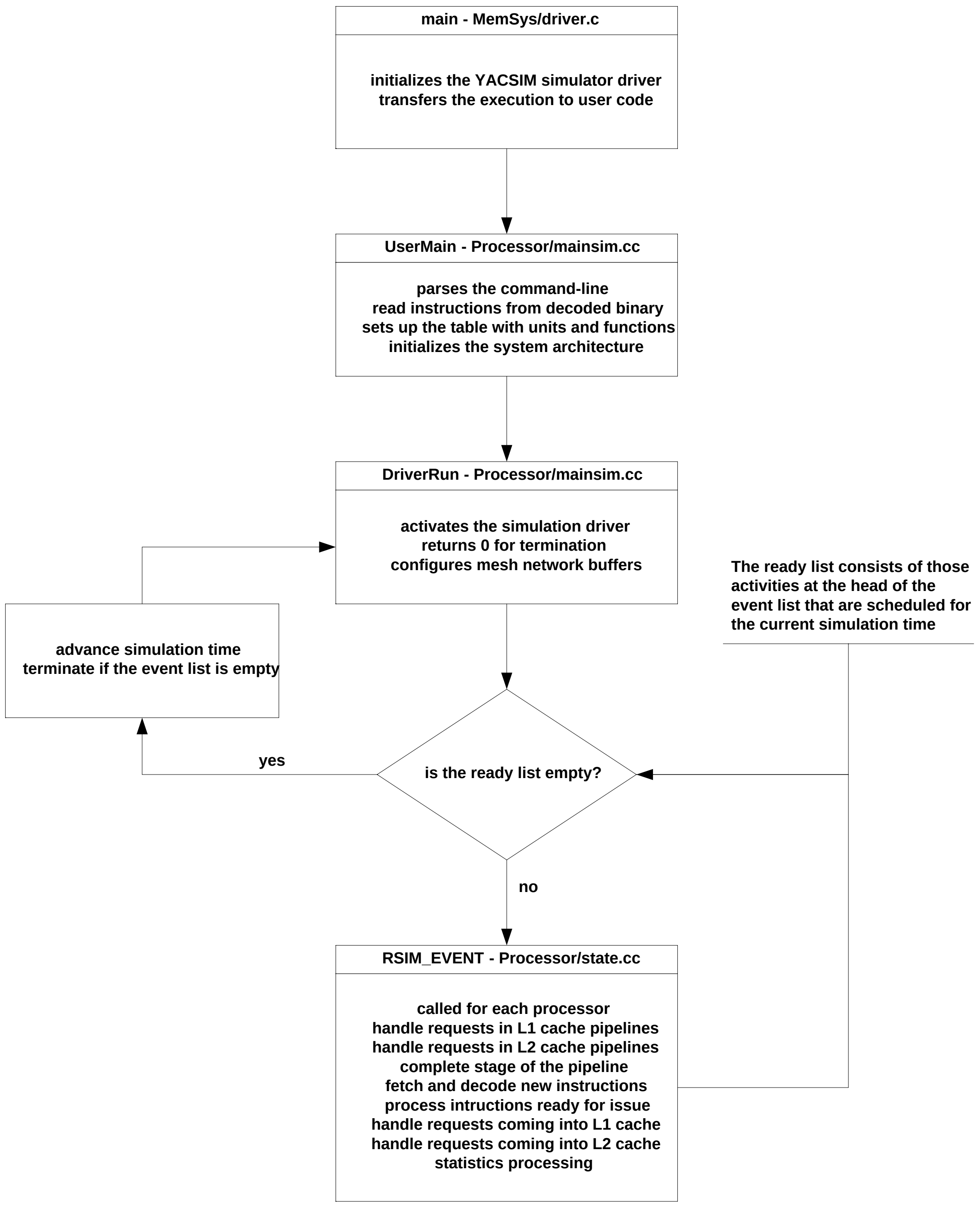

main - MemSys/driver.c
initializes the YACSIM simulator driver
transfers the execution to user code
UserMain - Processor/mainsim.cc
parses the command-line
read instructions from decoded binary
sets up the table with units and functions
initializes the system architecture
DriverRun - Processor/mainsim.cc
activates the simulation driver
returns 0 for termination
configures mesh network buffers
The ready list consists of those activities at the head of the event list that are scheduled for the current simulation time
advance simulation time
terminate if the event list is empty
yes
is the ready list empty?
no
RSIM_EVENT - Processor/state.cc
called for each processor
handle requests in L1 cache pipelines
handle requests in L2 cache pipelines
complete stage of the pipeline
fetch and decode new instructions
process intructions ready for issue
handle requests coming into L1 cache
handle requests coming into L2 cache
statistics processing

Figure 37: Execution flowchart of the simulator

As depicted in Figure 38, the `RSIM_EVENT` function simulates processor and cache operation. As it is likely that both processor and caches have nearly continuous activity, `RSIM_EVENT` is scheduled every cycle. However, in order to avoid non-deterministic behavior this function is scheduled to occur with an offset of 0.5 with respect to the processor cycles. As a consequence, `RSIM_EVENT` is executed at the midpoint between subsequent processor cycles in the simulation timeline.

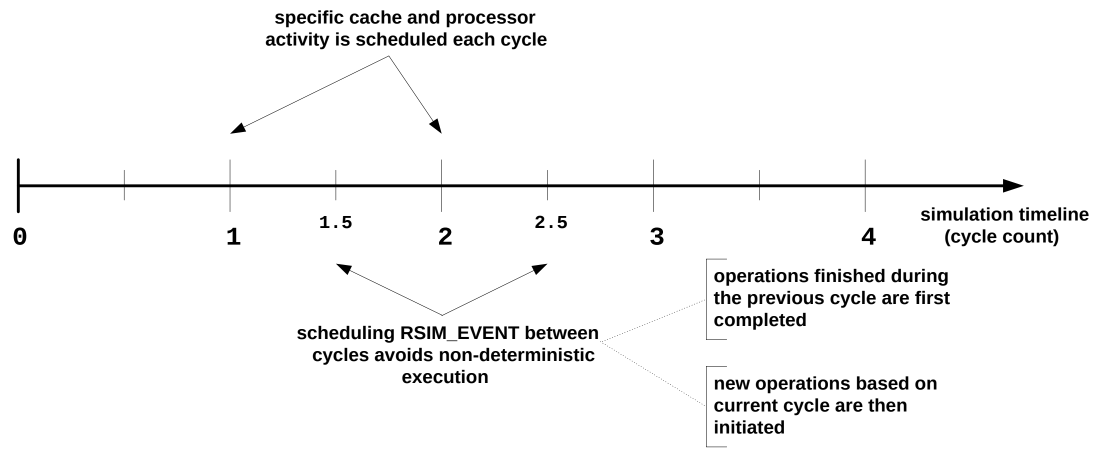

specific cache and processor activity is scheduled each cycle
0
1
1.5
2
2.5
3
4
simulation timeline (cycle count)
scheduling RSIM_EVENT between cycles avoids non-deterministic execution
operations finished during the previous cycle are first completed
new operations based on current cycle are then initiated

Figure 38: `RSIM_EVENT` scheduling

Figure 39 depicts instruction lifetime stages and the operations performed in each of these stages.

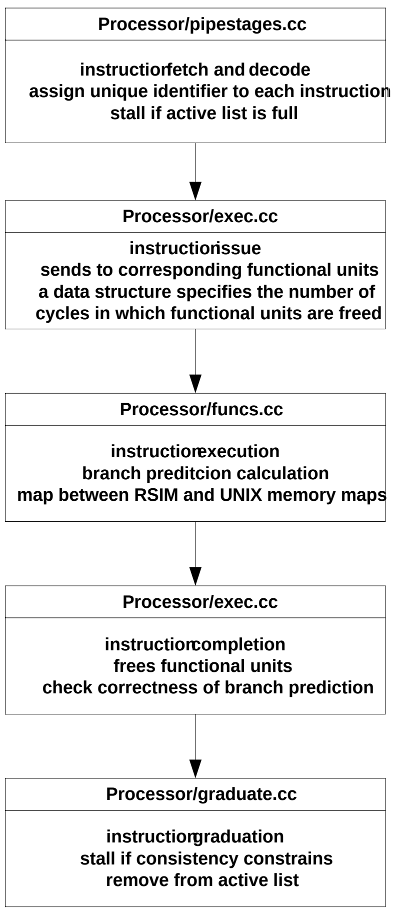

Processor/pipestages.cc
instruction fetch and decode
assign unique identifier to each instruction
stall if active list is full
Processor/exec.cc
instruction issue
sends to corresponding functional units
a data structure specifies the number of cycles in which functional units are freed
Processor/funcs.cc
instruction execution
branch preditcion calculation
map between RSIM and UNIX memory maps
Processor/exec.cc
instruction completion
frees functional units
check correctness of branch prediction
Processor/graduate.cc
instruction graduation
stall if consistency constrains
remove from active list

Figure 39: Instruction lifetime stages

3.5 Simulation results

A simple benchmark application that makes use of the set of fine-grain operations implemented by the simulation platform has been developed. The application core is a loop in which a node issues a trapping altering store and the rest of processors perform a non-altering load in parallel. Both the number of nodes and number of iterations are customizable by command-line parameters.

The operation of the fine-grained version of the application is defined below in the form of pseudo-code (complete source code is presented in Appendix B).

```
parse_command_line_parameters();    // Configure number of nodes and iterations.
allocate_shared_memory();           // To be used for the shared data array.
turn_on_memory_system_simulation(); // For collection of statistics.
create_processes();                 // Each process is run on a different node.
LOOP {                              // For the specified number of iterations.
   if (process_id != 0) {           // Processes other than the main process
      read_synchronized_var();      // perform a waiting non-altering read.
   } else {                         // The main process performs a trapping
      write_synchronized_var();     // altering write.
   }
}
```

A coarse-grained version of the same application has been implemented using barriers (see the complete source code in Appendix C). Its operation in the form of pseudo-code is detailed here for reference.

```
parse_command_line_parameters();    // Configure number of nodes and iterations.
allocate_shared_memory();           // To be used for the shared data array.
initialize_barrier(barrier);        // Rendez-vous point for all the processes.
turn_on_memory_system_simulation(); // For collection of statistics.
create_processes();                 // Each process is run on a different node.

LOOP {                              // For the specified number of iterations.
   if (process_id != 0) {           // Processes other than the main process
      wait_at_barrier(barrier);     // wait for the write to complete and
      read_shared_var();            // perform an ordinary read.
   } else {
      write_shared_var();
      wait_at_barrier(barrier);
   }
}
```

The results of a series of experiments are graphically depicted in Figure 40[1]. While the left plot shows execution times for different machine sizes, the right plot compares execution times for various problem sizes (i.e. number of iterations). Diverse components of execution time are distinguished by a different shade in each bar of these plots. Table 12 shows a tabulated version of the simulation results for a constant number of iterations, while Table 13 shows the same results for a fixed number of processing nodes.

As seen in Figure 40 (a) and Table 12, the execution time of the fine-grained version increases linearly with the problem size. Additionally, as the number of nodes decreases, the execution time slightly degrades. Effectively, the more nodes take part in a synchronization operation, the higher is the completion time. Both the cost of storage required for synchronization data and the traffic caused by these operations in the mesh network increase with the number of nodes. It is also important to observe that the most significant part of the execution time is due to cache and remote memory accesses.

The results, presented in Figure 40 (b) and Table 13, show that the dependence between the execution time and the number of nodes is remarkably higher in the coarse-grained version, which do not make use of a cache coherence protocol integrated with synchronization operations. This is because the overhead imposed by barriers is notably higher and so is its relevance in comparison with the total execution time. As a consequence, as the number of nodes increases, the execution time rises faster in the coarse-grained version. Note also that the accumulated times at the FPU stall and busy states are insignificant in comparison to the times spent for remote accesses and barrier synchronization.

1. The results have been derived from the statistics collection utilities distributed with RSIM.

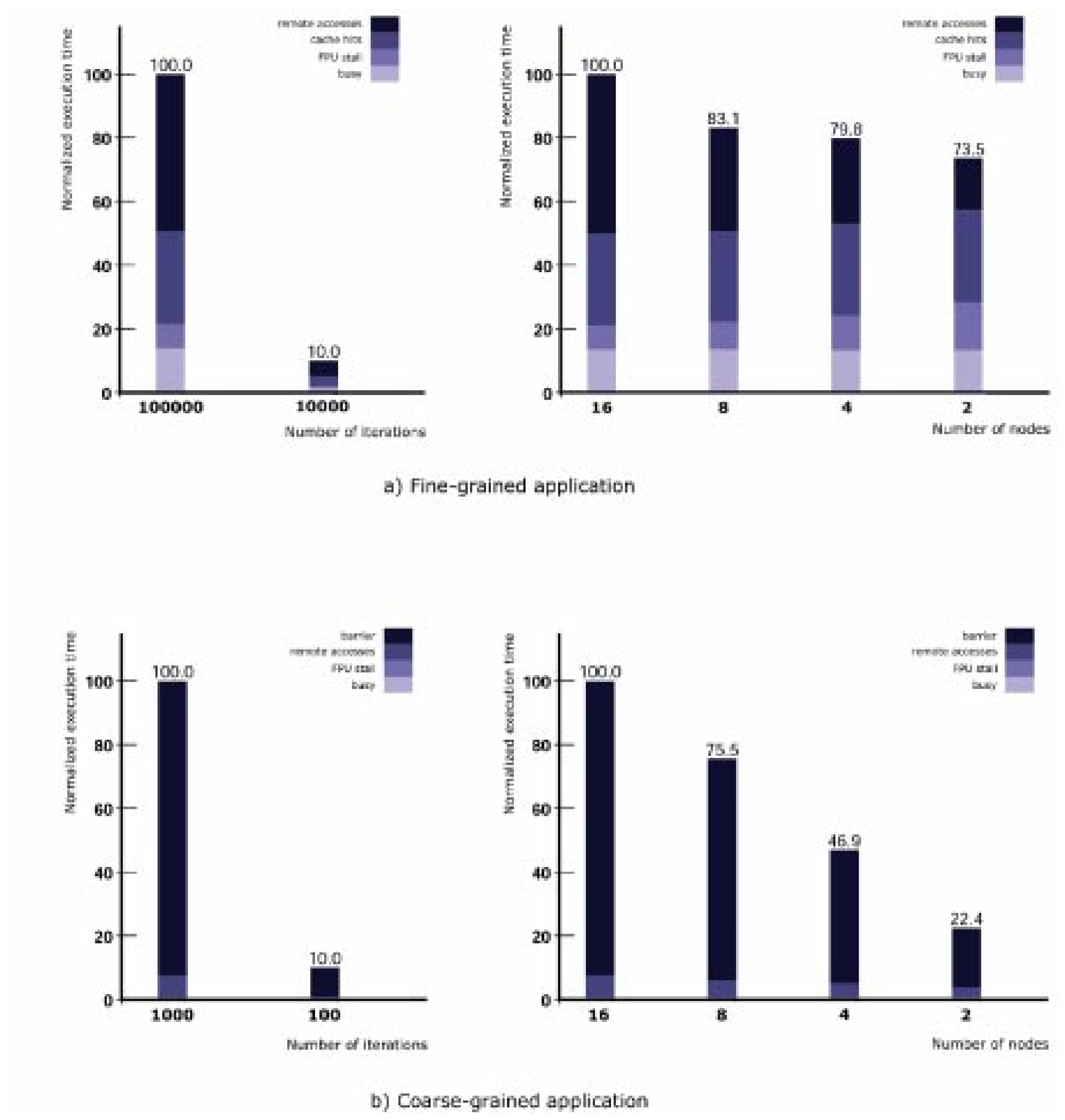


Figure 40: Normalized execution time for different machine and problem sizes

Table 12: Execution times (in cycles) for 1.000 iterations

Number of processors	Fine-grained version	Coarse-grained version
16	32176	2455349
8	26665	1845747
4	25983	1150828
2	24822	548853

Table 13: Execution times (in cycles) for 16 nodes

Number of iterations	Fine-grained version	Coarse-grained version
100000	3002126	-
10000	302068	-
1000	32176	2455349
100	4918	246280

Conclusions

Fine-grain synchronization is a valuable mechanism for speeding up the execution of parallel algorithms by avoiding false data dependencies and unnecessary process waiting. However, the implementation of fine-grain synchronization introduces additional complexity at both hardware and software system components.

A novel architecture with support for fine-grain synchronization at the cache coherence level is introduced. We propose a model that can be efficiently implemented in modern multiprocessors. The hardware overhead required by this architecture is not expected to be excessive.

Coherence protocols with support for fine-grain synchronization have been systematically described for both bus-based and directory-based multiprocessors. This work includes as well description of the rules for management and resuming pending requests, which is a key issue for the correct operation of the presented architecture.

Although it has not been completely developed yet, the simulation platform has been tested with a sample application making use of a small set of conditional operations. A coarse-grained version of the same application has been written and its simulation results compared to those of the fine-grained version, showing the performance improvements provided by the latter. These preliminary results verify the worthiness of implementing fine-grain synchronization at the cache coherence level.

Future work

Some features such as sophisticated management of pending requests have been specified but not yet source coded. Additionally, further debugging of the simulation platform is required[1]. This will not only verify the correct functioning of the protocols, but also evaluate design options that were taken during the specification process. Protocol verification with automatic verifier tools is also desirable.

Further simulation is required in order to obtain more precise quantitative data related with the performance of the proposed set of synchronization memory operations. In particular, the statistics collection functions implemented in the simulator platform should be modified so that the cost of storage required for synchronization data and the latency of fine-grain synchronization operations can be measured and easily compared with traditional synchronization mechanisms. Other important parameters to be measured are extra traffic caused by these operations and saturation that may be present at different levels of the memory hierarchy.

Extending a standard C compiler in order to support the complete set of synchronizing memory instructions would greatly simplify the steps required to perform a single simulation. An alternative is to extend the RSIM predecoder so that it recognizes Alewife binaries, which would eliminate the necessity of writing applications from scratch for making use of synchronizing operations. Note however that Alewife doesn't make use of the full set of proposed instructions.

Another pending task is to implement the full set of synchronizing instructions under RSIM. The evaluation of different coherence protocols other than MESI would also be very valuable, as well as developing extensive statistics collection in order to understand the tradeoffs involved in the proposed architectures. The implementation of a power estimation algorithm is also an appealing task, as considerable source code can be reused from already existing energy estimation tools.

Acknowledgments

This work is the result of the collaboration between the Department of Microelectronics and Information Technology (IMIT) of Royal Institute of Technology (KTH) with the Electrical and Computer Engineering Department at the University of Massachusetts (UMASS). We would like to thank Diganta Roychowdhury and Raksit Ashok, students at ECED UMASS, for their perceptive comments and valuable material.

A special thanks to the former members of the MIT Alewife Project Donald Yeung and Matthew Frank for providing excellent, incisive advise and fast response despite the huge amount of daily mail they have to answer.

1. The source code of the simulator has also been compiled with gcc instead of SUN's cc, allowing thus to debug under gdb, which provides many more debugging features than dbx.

Appendix A Preparing binaries for simulation

This Appendix describes in detail how to prepare a binary that makes use of fine-grain synchronization to be used with the extended RSIM simulator, supposing that we start from the following C source code. Note that the synchronized store is marked by a previous write to the ASI register, which is likewise performed by a call to the `asm` function, as shown below:

```
int sVar; /* synchronized variable */

  /* The values of ASI for synchronizing operations are:
     ASI_UE   0x91
     ASI_FF   0x96
     ASI_EF_T 0x9d
  */
  asm("wr   %g0, 0x9d, %asi");         /* WRASI instruction */

  sVar = 5;                            /* synchronized store (STWA_EFT) */
  /* The complete assembler sequence looks like this:
     wr   %g0, 0x9d, %asi
     mov  xxx, %o0                !xxx is the data to be stored
     stwa %o0, [%fp - yyy], %asi  !yyy is an appropriate offset
  */
}
```

In order to get an ordinary SPARC binary, SUN's `cc` compiler has to be invoked with the `xarch=v8plusa` option. Otherwise the store to the ASI register will not be recognized as a valid instruction. A binary is obtained with:

```
cc -xarch=v8plusa synch.c -o synch
```

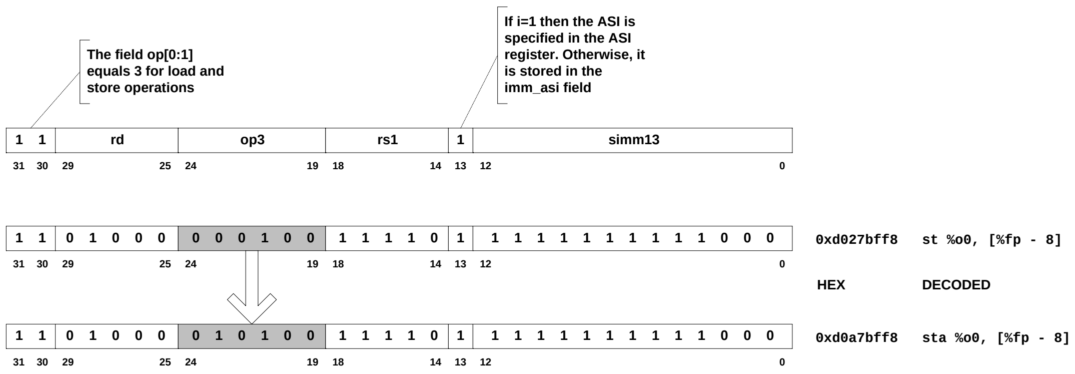
A disassembler is now used to calculate the file offset in which the store is located. The op-code of this store will be changed so that it is marked as synchronized. It is essential for the disassembler to support the SPARC32+ instruction set. The relevant disassembled output is listed below. Some instruction fields may vary depending on the particular system.

```
ADDRESS INSTRUCTION DECODED

0x000107f0 0x87826000 wr  %g0, 157, %asi
0x000107f4 0x90102005 mov 5, %o0
0x000107f8 0xd027bff8 st %o0, [%fp ñ 8]
0x000107fc 0x81c7e008 ret
```

As it is preceded by a `WR` instruction to the ASI register, it is straightforward to find the store whose opcode needs to be modified. The detailed instruction format of this store is shown in the following figure, in which it is also depicted the field to be changed so that the store is labelled as synchronized. This change can be easily made with a standard hexadecimal editor (`khexedit` has been used in this study).

As deduced from Figure 41, in this example the byte at offset `0x000107f9` has to be changed from value `0x27` to value `0xa7`. The resulting binary can be then predecoded and used as the input of the simulation platform.



The field op[0:1] equals 3 for load and store operations
If i=1 then the ASI is specified in the ASI register. Otherwise, it is stored in the imm_asi field
1 1 | rd | op3 | rs1 | 1 | simm13
31 30 29 25 24 19 18 14 13 12 0
1 1 | 0 1 0 0 0 | 0 0 0 1 0 0 | 1 1 1 1 0 | 1 | 1 1 1 1 1 1 1 1 1 1 0 0 0
0xd027bff8 st %o0, [%fp - 8]
HEX DECODED
1 1 | 0 1 0 0 0 | 0 1 0 1 0 0 | 1 1 1 1 0 | 1 | 1 1 1 1 1 1 1 1 1 1 0 0 0
0xd0a7bff8 sta %o0, [%fp - 8]

Figure 41: Details on how to transform a standard store to a synchronized store

Table 14 shows the relevant values of the field op3 and the corresponding operation associated to those values [21].

Table 14: Relevant values of the op3 field

Op-code	Operation	op3 field (binary)
LDUW	Load Unsigned Word	00 0000
LDUWA	Load Unsigned Word from Alternate space	01 0000
STW	Store Word	00 0100
STWA	Store Word from Alternate space	01 0100

Appendix B Application source (fine-grained version)

This appendix contains the source code listing of the sample application used in order to test the set of fine-grained synchronized memory operations. The `MEMSYS_OFF` and `MEMSYS_ON` calls make the simulator ignore non-relevant initialization steps. Note also that after allocating shared memory space for a given variable with `shmalloc`, it is necessary to define the home node that owns this space by using the `AssociateAddrNode` function.

```
#include <rsim_apps.h>
#include <stdio.h>
#include <stdlib.h>

/* #define __sparc_v9__ */

int NUM_PROCS  = 1; /* number of processors */
int ITERATIONS = 1; /* number of iterations */
int DEBUG      = 0; /* print debugging info */

int *sVar_;        /* shared array of size ITERATIONS */
int proc_id;       /* private variable */
int phase;         /* private variable */
extern char *optarg;

main(int argc, char **argv) {
  int c, i, j, dummy;

  MEMSYS_OFF; /* turn off detailed simulation for initialization */

  while ((c = getopt(argc, argv, "p:i:d")) != -1)
    switch (c) {
    case 'p':
      NUM_PROCS = atoi(optarg);
      break;
    case 'i':
      ITERATIONS = atoi(optarg);
      break;
    case 'd':
      DEBUG = 1;
      break;
    case 'h':
    default:
      fprintf(stdout, "SYNCH - OPTIONS\n");
      fprintf(stdout, "\tp - Number of processors\n");
      fprintf(stdout, "\ti - Number of iterations\n");
      fprintf(stdout, "\td - Print debugging info\n");
      fprintf(stdout, "\th - Help\n");
      return;
    }

  sVar_ = (int*) shmalloc(ITERATIONS * sizeof(int));
  AssociateAddrNode(sVar_, sVar_ + ITERATIONS, 0, "sVar");

  if (sVar_ == NULL) {
    fprintf(stdout, "Unable to malloc shared region\n");
    exit(-1);
  }

  if (DEBUG)
    fprintf(stdout, "Running with %d processors and %d interations...\n\n",
       NUM_PROCS, ITERATIONS);
  MEMSYS_ON;

  proc_id = 0;
  for (i=0; i<NUM_PROCS-1; i++) {
    if (fork() == 0) {
      proc_id = getpid();
      break;
    }
  }

  if (proc_id == 0) {
    StatReportAll();
    StatClearAll();
  }
  endphase();
  newphase(++phase);         /* beginning of new phase */

  for (j=0; j<ITERATIONS; j++) {
    if (proc_id == 0) {      /* the main thread stores (STWA_EFT) the value */
      /* Values of ASI for synchronizing operations:
    ASI_UE   0x91
    ASI_FF   0x96
    ASI_EF_T 0x9d
      */
      asm("wr   %g0, 0x9d, %asi");
      sVar_[j] = 9;          /* synchronized store (STWA_EFT) */

      if (DEBUG)
   fprintf(stdout, "%d: Stored value %d from sVar_[%d]\n",
      proc_id, sVar_[j], j);

      /* The complete assembler sequence looks like this:
    mov  0x9d, %o1
    wr   %o1, 0x0, %asi
    mov  xxx, %o0                !xxx is the data to be stored
    stwa %o0, [%fp - yyy], %asi  !yyy is an appropriate offset
      */
    } else { /* the rest of the threads try to LDWA_FF the value */
      /* Values of ASI for synchronizing operations:
    ASI_UE   0x91
    ASI_FF   0x96
    ASI_EF_T 0x9d
      */
      asm("wr   %g0, 0x96, %asi");
      dummy = sVar_[j];        /* synchronized load (LDWA_FF) and store to a
            standard dummy variable */
      if (DEBUG)
   fprintf(stdout, "%d: Read value %d from sVar_[%d]\n",
      proc_id, sVar_[j], j);
    }
   }

  if (DEBUG)
    fprintf(stdout, "\nProcessor %d about to finish!\n\n", proc_id);

  exit(0);                 /* completed successfuly */
}
```

Appendix C Application source (coarse-grained version)

Below is the source code listing of the sample application implementing the same functionality of the fine-grained version by using barriers. Calls to the directives `MEMSYS_OFF` and `MEMSYS_ON` are used as in the fine-grained version (Appendix B). Barriers are initialized and activated by invoking `TreeBarInit` and `TREEBAR`, respectively.

```
#include <rsim_apps.h>
#include <stdio.h>
#include <stdlib.h>

/* #define __sparc_v9__ */

int NUM_PROCS  = 1; /* number of processors */
int ITERATIONS = 1; /* number of iterations */
int DEBUG      = 0; /* print debugging info */

int *sVar_;        /* shared array of size ITERATIONS */
TreeBar barrier;   /* tree barrier */
int proc_id;       /* private variable */
int phase;         /* private variable */
extern char *optarg;

main(int argc, char **argv) {
  int c, i, j, dummy;

  MEMSYS_OFF; /* turn off detailed simulation for initialization */

  while ((c = getopt(argc, argv, "p:i:d")) != -1)
    switch (c) {
    case 'p':
      NUM_PROCS = atoi(optarg);
      break;
    case 'i':
      ITERATIONS = atoi(optarg);
      break;
    case 'd':
      DEBUG = 1;
      break;
    case 'h':
    default:
      fprintf(stdout, "SYNCH - OPTIONS\n");
      fprintf(stdout, "\tp - Number of processors\n");
      fprintf(stdout, "\ti - Number of iterations\n");
      fprintf(stdout, "\td - Print debugging info\n");
      fprintf(stdout, "\th - Help\n");
      return;
    }

  sVar_ = (int*) shmalloc(ITERATIONS * sizeof(int));
  AssociateAddrNode(sVar_, sVar_ + ITERATIONS, 0, "sVar");

  if (sVar_ == NULL) {
    fprintf(stdout, "Unable to malloc shared region\n");
    exit(-1);
  }

  TreeBarInit(&barrier, NUM_PROCS); /* initialize tree barrier */

  if (DEBUG)
    fprintf(stdout, "Running with %d processors and %d interations...\n\n",
       NUM_PROCS, ITERATIONS);
  MEMSYS_ON;

  proc_id = 0;
  for (i=0; i<NUM_PROCS-1; i++) {
    if (fork() == 0) {
      proc_id = getpid();
      break;
    }
  }

  if (proc_id == 0) {
    StatReportAll();
    StatClearAll();
  }
  endphase();
  newphase(++phase);         /* beginning of new phase */

  for (j=0; j<ITERATIONS; j++) {
    if (proc_id == 0) {      /* the main thread stores the value */
      sVar_[j] = 9;          /* ordinary store */
      if (DEBUG)
   fprintf(stdout, "%d: Stored value %d from sVar_[%d]\n",
      proc_id, sVar_[j], j);
      TREEBAR(&barrier, proc_id);
    } else { /* the rest of the threads try to load the value */
      TREEBAR(&barrier, proc_id);
      dummy = sVar_[j];        /* ordinary load and store to dummy variable */
      if (DEBUG)
   fprintf(stdout, "%d: Read value %d from sVar_[%d]\n",
      proc_id, sVar_[j], j);
    }
  }

  if (DEBUG)
    fprintf(stdout, "\nProcessor %d about to finish!\n\n", proc_id);

  exit(0);                 /* completed successfuly */
}
```

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.; Beng-Hong Lim; Kranz, D. and Kubiatowicz, J.: “APRIL: A Processor Architecture for Multiprocessing”, Laboratory for Computer Science, Massachusetts Institute of Technology, 1990

[3] 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, Massachussets Institute of Technology, 1999

[4] 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

[5] 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, Massachussets Institute of Technology, 1993

[6] Ang, B.S. et. al.: “StarT the Next Generation: Integrating Global Caches and Dataflow Architecture”, in Advanced Topics in Dataflow Computing and Multithreading, IEEE Press, 1995

[7] Boughton, R.D.: “Artic Routing Chip”, in Parallel Computer Routing and Communications Proceedings of the First International Workshop, PCRW ´94, volume 853 of Lecture Notes in Computer Science, pages 310-317, May 1994

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

[9] Chiou, D.; Ang, B.S.; Arvind et. al.: “StarT-NG: Delivering Seamless Parallel Computing”, Euro-Par ´95, August 1995

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

[11] Hoare, C.A.R.: “Monitors: An Operating System Structuring Concept”, Communications of the ACM, October 1974, pages 549-557

[12] Hughes, C.J.; Pai, V.S.; Ranganathan, P. and Adve, S.V.: “Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors”, IEEE Computer, February 2002, pp 44-49

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

[14] 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)

[15] 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

[16] 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

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

[18] Lim, B.H. and Agarwal, A.: “Reactive synchronization Algorithms for Multiprocessors”, Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, 1994

[19] Pai, V.S.; Ranganathan, P. and Adve, S.V.: “RSIM Reference Manual”, Department of Electrical and Computer Engineering, Rice University, version 1.0, August 1997

[20] 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

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

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

[23] Xiaowei, Shen: “Implementing Global Cache Coherence In *T-NG”, MSc. Thesis at the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1995

[24] 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