KUNGL TEKNISKA HÖGSKOLAN
Royal Institute of Technology

Efficient Fine Grained Synchronization Support Using Full/Empty Tagged Shared Memory and Cache Coherency

Vladimir Vlassov and Csaba Andras Moritz

Department of Teleinformatics

KUNGL TEKNISKA HÖGSKOLAN
Royal Institute of Technology

Efficient Fine Grained Synchronization Support Using Full/Empty Tagged Shared Memory and Cache Coherency

Vladimir Vlassov[*] and Csaba Andras Moritz[**]

[*] Department of Teleinformatics
Royal Institute of Technology (KTH)
Stockholm, Sweden

[**] Department of Elect. and Computer Engineering
University of Massachusetts (UMASS), Amherst,
Amherst, MA, USA

TRITA–IT–00:04
ISSN 1103–534X
ISRN KTH/IT/R—00/04—SE

Department of Teleinformatics

Abstract

Performance results of machines with fine-grain synchronization on individual lock-free data items (e.g., words), such as the MIT Alewife multiprocessor, illustrate the benefits of supporting fine-grain synchronization. The performance benefits are primarily the result of allowing a dataflow style of computation in programming models, and maximizing the exposed parallelism by minimizing the possibility of false dependencies caused by coarse grained synchronization.

In this report we propose a new efficient way to support fine grained synchronization mechanisms on multiprocessors. We propose to design a full/empty tagged memory hierarchy with aggressive hardware support for fine grained synchronization that is embedded in the cache coherency mechanism of an SMP or a NUMA multiprocessor, or a single-chip multiprocessor. We believe that handling synchronization and coherence together can provide a more efficient execution, reducing the occupancy in the memory controllers and the network bandwidth consumed by protocol messages.

The fine-grain synchronization mechanism can be implemented with a full/empty tagged shared memory where a full/empty bit is associated with each word. Such a memory is accessed by special memory operations (loads, stores and swaps) that may depend on the full/empty state of the target location and can alter the state. A synchronization fault (we call it a state miss) occurs when the required state of the location is not met. Our objective is to improve the performance of the full/empty synchronization mechanism such as implemented in the MIT Alewife machine, by integrating a cache coherency mechanism with the full/empty synchronization.

To achieve this, we propose to handle synchronization faults in a similar way as cache misses in a lockup-free cache. This allows implementation of non-faulting (non-trapping) full/empty memory operations. In our design, we assume that a full/empty memory operation suspends on a synchronization miss (by analogy to a cache miss) waiting in the memory while the miss is resolved. An out-of-order processor and a lockup-free miss-under-miss cache organization allow to tolerate most of the synchronization miss latency. There are several design issues to be considered for a full/empty tagged memory, such as how to hide the state miss latency and how to prevent saturation of the memory hierarchy with unresolved state misses. We also propose to have architecture level support for fine grained synchronization by associating a full/empty bit with each processor register. This may allow to achieve efficient thread synchronization on the register rather than on the memory access level.

1 Introduction

Parallel and distributed computing has emerged to be one of the promising developments that can extend human capabilities in many fields of activities, such as numeric simulation and modeling of physical phenomena and complex systems, and different forms of information processing on the Internet.

Performance of parallel and distributed platforms (including cluster of workstations) continues to improve due to technological advances (i.e. improvement in logic density and clock frequency), due to efficient architectural solutions that translate the potential of technology into performance and capability, and due to efficient software abstractions for parallel applications. Computer architects and compiler writers continuously try to exploit more of the parallelism available in programs at different granularity levels (i.e. instructions, threads, and processes) in their quest for higher performance. Some of these techniques such as control-flow speculation, memory-dependence and value speculation, and fine grained synchronization support, are designed to maximize the available program parallelism.

Synchronization of parallel processes and threads is an important mechanism in parallel and concurrent programming. Synchronization insures correctness of parallel execution by enforcing true data dependencies and timing constraints. In a parallel programming environment based on a shared-memory programming model, synchronization is provided either as explicit user-level synchronization primitives such as locks and barriers, or implicitly synchronized data structures such as lock-able L-structures [12, 9] and write-once I-structures [5]. The performance benefits of supporting fine grained synchronization are primarily the result of allowing a dataflow style of computation in programming models, and maximizing the exposed parallelism by minimizing the possibility of false dependencies caused by coarse grained synchronization.

In this project we propose a new efficient way to support fine grained synchronization mechanisms on multiprocessors. We propose to design a full/empty tagged memory hierarchy with aggressive hardware support for fine grained synchronization that is embedded in the cache coherency mechanism of an SMP or a NUMA multiprocessor, or a single-chip multiprocessor. We believe that handling synchronization and coherence together can provide a more efficient execution, reducing the occupancy in the memory controllers and the network bandwidth consumed by protocol messages.

2 Research Overview

A synchronization mechanism can be classified as (i) coarse-grain synchronization where a synchronizing variable is associated with multiple shared locations and is used to control the order of accesses to these locations, or (ii) fine-grain synchronization where a synchronizing variable is associated with a single word or a block of memory. Thus, the granularity of synchronization is measured in the amount of data that is communicated with the synchronization [12]. The synchronization overhead in a shared memory multiprocessor includes the following measures: Cost of storage required for synchronization data (locks, counters, full/empty bits), synchronization latency that is observed on execution of synchronization primitives by one or multiple processes simultaneously, and amount of traffic in the interconnect or amount of bus transactions in a bus-based multiprocessor caused by synchronization.

One of the advantages of coarse grain synchronization based on locks and barriers is ease of programming, being the reason why such type of synchronization is widely used in shared-memory multiprocessors. A number of software and hardware implementations and optimizations were proposed for efficient lock and barrier synchronization in shared memory multiprocessors [15], such as the followings: a test-test-and-set spinlock, a spinlock with exponential back-off, software queuing locks and cache-based queuing locks. As reported in [11], a cache-based queuing lock, called QOLB (Queue on Lock Bit), is shown be the most efficient coarse-grain synchronization mechanism among others evaluated in [11]. A semantic weakness of lock synchronization as a basic synchronization mechanism for parallel processing is that locking simultaneously provides both atomicity and exclusivity of shared memory accesses, while some of the data protected by lock may not require exclusive rights to access them. Such shared data are called lock-free. Thus, lock synchronization may expose false dependencies which cause degradation of parallelism. A pair of instructions load-linked and store-conditional (LL and SC) supports a split load-modify-store transaction where SC returns a value that deduces whether the transaction initiated by previous LL was performed atomically. These instructions can be found, for example, in instruction sets of the MIPS R10000 processor [27]. A similar functionality is achieved with the atomic compare-and-swap instruction introduced in the SPARC-V8 (V9) architecture [14] and used in combination with ordinary load. Coarse grain synchronization, such as locking or barriers, can expose false dependencies that cause degradation of parallelism. Other problems of lock synchronization are priority inversion and convoying when a process holding a lock is interrupted for some reasons (descheduled or pre-emptied).

Fine-grain synchronization minimizes the possibility of false dependencies caused by coarse grained synchronization, such as locks and barriers, and maximizes the exposed parallelism. Therefore, fine-grain synchronization improves the efficiency of parallel execution. A fine-grain synchronization mechanism can be implemented with a full/empty tagged shared memory (shortly FE-memory) where a full/empty bit is associated with each word. Such a memory is accessed by special memory operations (loads, stores and swaps) that may depend on the full/empty state of the target location and can alter the state. The fine grained synchronization allows the dataflow style of computation. HEP [19], Tera [4, 7], and the MITs Alewife machine [1] are examples of multiprocessors with full/empty bit synchronization. Performance results of the MITs Alewife multiprocessor presented in [28] illustrate the advantage of the fine grained synchronization on individual lock-free data items (e.g., words). The MITs Alewife machine is a CC-NUMA multiprocessor with a full-empty tagged distributed shared memory and hardware-supported block-multithreading. The MIT Alewifes processor Sparcle is an extension of an industry-standard SPARC v7 processor that is modified to support full/empty synchronization and multithreading partly in hardware and partly in software. Hardware support includes full/empty bits, special memory operations and a special full/empty trap. The software support includes trap handler routines for fast context switching and waiting for synchronization [13].

As illustrated in [28] for the Alewife machine, the fine-grain synchronization expressed with synchronizing data structures (J-structures) and supported at low-level with the full/empty word-level synchronization, outperforms the course-grain barrier synchronization. The experience reported in [28] shows that the bulk of the performance advantage due to increased parallelism is caused by expressing and utilizing synchronization at a fine granularity. Barriers impose false dependencies and thus inhibit parallelism because of unnecessary waiting. With fine-grain synchronization, unnecessary waiting is avoided because a consumer waits only for the data it needs.

In the MITs Alewife machine the full/empty synchronization is implemented on top of cache coherency, i.e. the synchronization protocol is a separate layer on top of the underlying cache coherence protocol. However, we believe that combining the fine-grain synchronization and cache coherency together can potentially produce more efficient execution. In Alewife, a synchronization failure that occurs when a requested full/empty state of a target location is not met, is treated as an error to be handled in software through a light-weight trap [2]. Handling synchronization faults in software has the advantage of flexibility though it is expensive. Coarse-grained multithreading supported in hardware is used in Alewife for hiding latency caused by synchronization failures.

A few other approaches to fine-grain synchronization exist in other research multiprocessors, such as the M-machine with full/empty tagged registers [10] and a simultaneous multithreaded (SMT) processor with hardware-based blocking locks described in [21]. Both mechanisms are proposed for efficient fine-grain synchronization of threads within a processor. However, these designs do not provide the fine-grain synchronization across multiple processors.

Our intention is to improve the performance of a fine-grain synchronization mechanism such as the full/empty synchronization in the shared memory implemented in the MIT Alewife machine and the full/empty register-level synchronization within a processor proposed for the M-machine. We believe, that both mechanisms (within a processor and across processors, in the shared memory) must be designed and considered together in a consistent way. Our first target is hardware support for the full/empty synchronization in the shared memory, i.e. across processors rather than within a processor. To achieve this we intend to design and to evaluate a full/empty memory hierarchy where a cache coherency mechanism is integrated with the full/empty synchronization. Combining synchronization and coherence together is the major feature that differentiates our design from previous. In particular, we propose to handle synchronization faults in a similar way as cache misses in a lockup-free cache. This allows implementation of non-faulting (non-trapping) full/empty memory operations. In our design, we assume that a full/empty memory operation suspends on a synchronization miss (by analogy to a cache miss) waiting in the memory while the miss is resolved. An out-of-order processor and a lockup-free miss-under-miss cache organization allow to tolerate most of the synchronization miss latency. We also propose not to send data in the memory hierarchy on a cache miss if the requested state of a location is not met. We believe, that this optimization of a coherence protocol allows to reduce the bandwidth demand on system interconnect significantly.

In recent super-scalar RISC microprocessors, the parallel execution at the instruction level is both control-driven (in-order fetch and in-order completion), and data-driven (out-of-order issue and out-of-order execution). As the FE-synchronization facilitates a dataflow style of parallel execution in a shared memory model, this type of synchronization, in our view, helps to reduce the gap between the data-flow style of the instruction-level parallelism and the control-flow style of high-level programmer-oriented parallelism when synchronized with locks and barriers in a "user-controlled" parallel programming environment. FE-synchronization can also be effectively exploited in parallel compilers.

3 EDA: Extended Dataflow Actor Model

This section contains some of the preliminary work done by the authors at the Royal Institute of Technology (KTH), Stockholm, Sweden. We briefly present a parallel computation model, Extended Dataflow Actor (EDA), that has been developed at KTH during last few years [24, 16, 25, 26, 20, 3, 22, 17, 23].

The EDA model was mainly inspired from the data flow model Actors. Its origin is the Ph D work by Handong Wu [24]. The goal of the development and implementation of the EDA model was to provide a flexible, effective and relatively simple model of thread (process) communication and synchronization via synchronizing shared memory. In EDA, a synchronizing shared memory is achieved by a binary (full/empty) state associated with each shared memory location. The investigator has participated in defining the EDA language constructs enabling implicit synchronization, and implemented a mapping tool for EDA programs [17].

The EDA model specifies special synchronization types for shared variables similar to data structures with special accessors, such as I-structures [5] and M-structures [6], used in a programming environment for the MIT Monsoon dataflow machine synchronization in data parallelism programmed for the MIT Alewife shared-memory multiprocessor [1].

In EDA, shared variables are of three different synchronization types, I-data, X-data, and S-data, which all impose different constraints in the way accesses may be performed. I-data are used for enforcing data dependency: a read operation on an empty variable will lead to suspension, and assignment is allowed only once. A write operation on a full I-data variable will simply be discarded, thus supporting a kind of OR-parallelism. X-data are used for mutual exclusion and synchronous communication: reads and writes must be performed in a strictly alternating sequence. A thread attempting an access violating this order will be delayed until another thread has changed the state of the accessed variable. S-data allow stream communication. A writing thread (process) can assign successive values to an S-type variable; these will be enqueued and available for subsequent read operations. Each read removes one value from the stream. Threads accessing a stream need not be suspended, except in the case of reading from an empty stream. Figure 1 depicts categories of fetch and store operations defined in EDA.

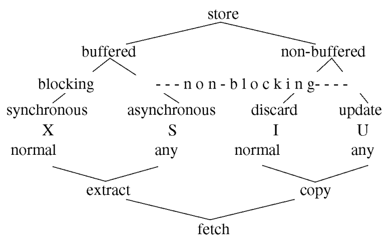

Figure 1: Categorization of EDA Operations

The EDA model was implemented on top of PVM as a parallel programming environment for multi-threaded computation [3] and as an extension library for PVM that allows storing, reading and extracting PVM messages from virtual shared cells [22, 23]. The mEDA model and its PVM implementation (mEDA-2 library) was used in developing of a distributed object-oriented environment NUTS system of collaborative NUT processes running on PVM, for exchanging classes, scripts and objects among NUT processes.

The experience of implementating the EDA model and developing EDA based parallel programs and its optimizations for performance, allowed us to understand the opportunities for efficient support of fine grained synchronization mechanisms at the architecture, at the language semantics level, and at the compiler level. The main objective of our project is to materialize these experiences in order to provide efficient fine grained synchronization support in future parallel computer systems.

4 Specifying a Full/Empty Tagged Memory

The main goal of this research is the design and evaluation of a full/empty tagged memory hierarchy with hardware support for fine-grain synchronization embedded in a cache coherency mechanism of an SMP or a NUMA multiprocessor, or a single-chip multiprocessor. Our objective is to develop an efficient way to support fine-grained synchronization in multiprocessors.

Hardware-supported fine-grain synchronization has been shown to outperform coarse-grained synchronization such as implemented with barriers [28]. Combining fine-grain synchronization and cache coherence together is the major feature that differentiates our design from previous approaches.

The MIT Alewife multiprocessor supports fine-grain synchronization in hardware [1]. It provides full/empty bits in memory and special synchronizing instructions with light-weight traps on synchronization faults. However, the full/empty synchronization is implemented on top of cache coherency. This causes unnecessary actions performed in the underlying layer of the coherence protocol in the case of a synchronization failure detected by synchronization protocol. The synchronization failure is handled in software and may require to maintain a queue of threads waiting for synchronization. Handling synchronization failures in software is flexible but very expensive, as a handling routine is on the critical path of the memory operation.

We believe that integrating the fine-grain synchronization with cache coherency potentially produces more efficient parallel execution. To achieve this, we propose, in particular, to handle synchronization faults in a similar way as cache misses in a lockup-free cache. This allows implementation of non-faulting (non-trapping) full/empty memory operations. In our design, we assume that a full/empty memory operation suspends on a synchronization miss (by analogy to a cache miss) waiting in the memory while the miss is resolved. In this way, a queue of waiting threads will be naturally maintained as a queue of outstanding state misses, by analogy to cache misses. Note that most of hardware complexity required by the lockup-free cache organization that allows multiple outstanding cache misses is already present in the memory system of modern shared-memory multiprocessors. An out-of-order processor and a lockup-free miss-under-miss cache organization allow to tolerate most of the synchronization miss latency.

There are several design issues to be considered for a full/empty tagged memory, such as how to hide the state miss latency and how to prevent saturation of the memory hierarchy with unresolved state misses. We intend to investigate how the full/empty shared memory can be combined with full/empty tagged registers of a processor. This combination allow to achieve efficient communication and synchronization of simultaneous threads on the registers rather than on the memory (cache) level.

This section describes how a full/empty memory hierarchy can be achieved. We start with the specification of operations that can be performed on the full/empty tagged memory. Then we introduce memory instructions which implement the specified operations and discuss how the instructions can be provided.

4.1 Full/Empty Memory Operations

Assume that in a shared memory each word has a full/empty bit (FE-bit) associated with it. We call such memory full/empty tagged memory or shortly FE-memory. The FE-bit indicates a binary state (FE-state) of the location: if the bit is set the location is full, otherwise the location is empty. The full state can be also interpreted as bound, defined, containing a "top" (meaningful) value from a flat domain. The empty state can interpreted as unbounded, undefined, containing a "bottom" (meaningless or "empty") value. FE-memory access operations that may depend on the FE-state and can alter the state can be provided for fine-grain data-flow-like synchronization performed at the FE-memory.

In general, the FE-memory can be decomposed into three logical parts: (i) a data memory, DM, that holds data, (ii) a state memory, SM, that holds FE-bits, and (iii) a state miss memory, SMM that holds pending access requests (state misses discussed later in this chapter). A state of a FE-memory location can be represented by a triplet d, s, w where d is a value stored in the DM, s is the value of the full/empty bit associated with the location and stored in SM, and w is a list of postponed memory requests (state misses) awaiting for the required state of the target location.

Figure 2 illustrates a possible logical organization of a full/empty memory and a full/empty cache in a bus-based shared-memory multiprocessor (only one node is shown). We assume that state misses can be treated in the same way as cache misses and the information that keeps track of outstanding misses (state and cache) is stored in miss state holding registers. Note that full/empty bits can be stored together with tags in the tag-directory of the FE-cache.

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

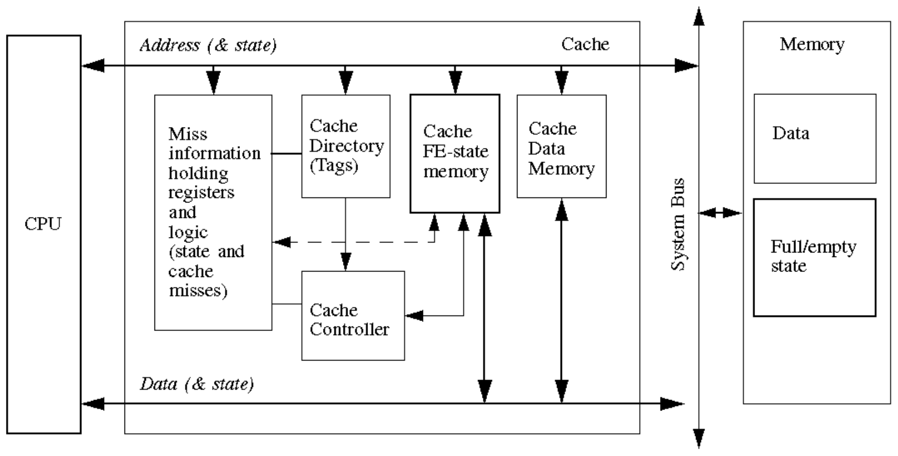

Figure 2: Organization of a Full/Empty Memory and a Full/Empty Cache

FE-memory operations can be further categorized according to how the result of an operation depends on the original full/empty state of the location (unconditional versus conditional), how a conditional operation behaves when the required state of the location is not met (non-waiting versus waiting), and how a non-waiting conditional operation treats a state miss (non-faulting versus faulting). The categories of the FE-memory operations are depicted in Figure 3.

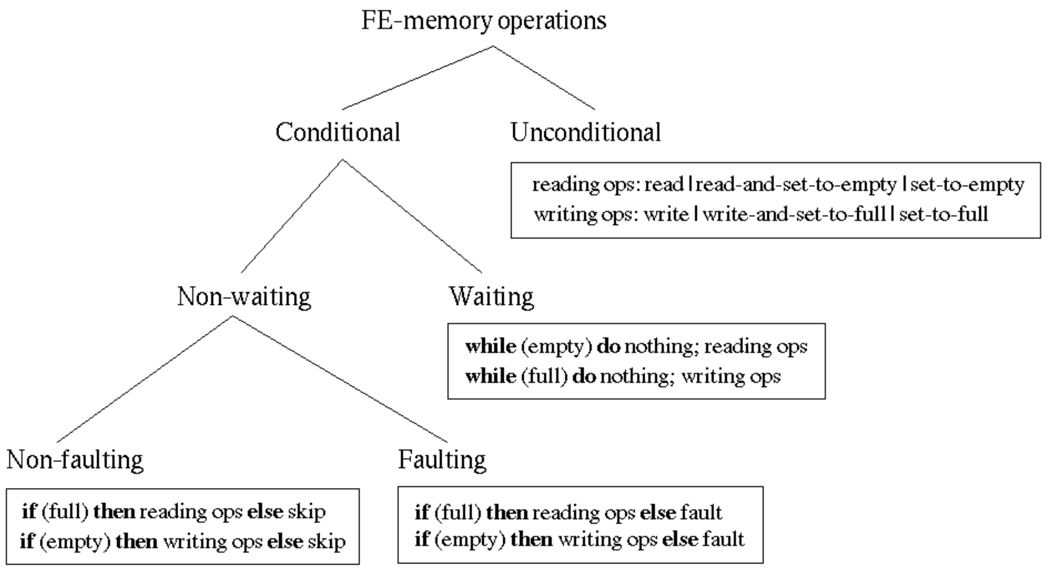
FE-memory operations
Conditional
Unconditional
reading ops: read | read-and-set-to-empty | set-to-empty
writing ops: write | write-and-set-to-full | set-to-full
Non-waiting
Waiting
while (empty) do nothing; reading ops
while (full) do nothing; writing ops
Non-faulting
if (full) then reading ops else skip
if (empty) then writing ops else skip
Faulting
if (full) then reading ops else fault
if (empty) then writing ops else fault

Figure 3: Categorization of FE-Memory Operations

Conditional versus unconditional. A memory operation is *conditional* if its result depends on a state of a target location. We assume that a conditional read returns a value of the location if and only if the location is full. A conditional write updates the location if and only if the location is empty. A conditional set alters a state if the previous state is opposite to the state to be set. A conditional altering operation accesses a location and sets a new state for the location if and only if the previous state of location corresponds to the state required in the operation. For example, the conditional altering read returns a value of location and sets its state to empty, if and only if the location is full. An *unconditional* operation always succeeds regardless of the FE-state of the target location.

Waiting conditional versus non-waiting conditional. If the condition is not met for a conditional operation, the operation should not be allowed to succeed. We call such situation *state miss*. We introduce two types of conditional operations (non-waiting versus waiting) according to how operations behave on a state miss. A *non-waiting* operation is allowed to succeed on a state hit, but it is dropped when a state miss occurs. A *waiting* conditional operation is postponed in the memory until a state miss is resolved. This requires that the memory keeps track of outstanding state misses (suspended operations) in a way that is similar to keeping track of outstanding cache misses. An altering operation can cause a pending operation to wake up. In general, wakeups should not be on the critical path of the altering operation that causes them.

Faulting versus non-faulting. Non-waiting conditional operations can be further categorized according to how a state miss is treated (faulting versus non-faulting). A *faulting* operation treats a state miss as an exception (a synchronization fault) which has to be handled in software. The synchronization fault induces a trap, and a trap handler may either retry the operation immediately (spin) or switch to another context. A *non-faulting* conditional operation does not treat a state miss as an error and does not require the miss to be resolved. Such operation is dropped on a state miss. We assume that each of the FE-operations (Fig. 3) has a version with a side effect that returns an original value of the full/empty bit associated with the location (to be used as a full/empty condition code in the processor.

4.2 Implementation Issues

This section describes some of the most important implementation issues to be considered for a full/empty tagged memory, and their possible solution.

4.2.1 Storing FE-State

One FE-bit per 4 bytes of a memory word implies an overhead of only 3%. Efficient solution for storing FE-bits was proposed and implemented in the MIT Alewife distributed memory multiprocessor [1]: a vector of 4-full/empty-bits associated with four words in a memory block are stored in the coherence directory entry at memory and as an extra field in the cache tag when the block is cached. This way, a tag (directory) lookup includes tags match and inspection of full/empty bits. In data packets the full/empty bits are transmitted with the address (in the bottom four bits).

4.2.2 FE-Memory Instructions

The FE-memory operations can be implemented by extending a RISC processor architecture with special memory instructions, and by embedding the full/empty synchronization mechanism into the memory hierarchy. We assume that the FE-memory hierarchy of a shared-memory multiprocessor includes a coherent layer(s) of processor-private FE-caches between processors and the shared memory that can be either centralized or distributed.

Typically an FE-memory load instruction loads a shared location to the specified register, a store instruction updates the location by a content of the specified register. An altering FE-memory instruction can set a new state to the location. A version of a FE-memory instruction with a side effect sets a full/empty condition code to the original value of the full/empty bit of a target location at the time the instruction starts execution.

For example, the MIT Alewife's processor Sparcle is an extension of an industry-standard SPARC v7 processor, that provides colored integer load and store instructions for full/empty test-and-set operations in a distributed shared memory of the Alewife multiprocessor [2]. In the Alewife, each 32-bit data word of the shared memory is supplied with a full/empty bit. In our terminology (Fig. 3), Sparcle provides unconditional and conditional faulting (non-waiting) loads and stores with the side effect described above. In Sparcle, the integer synchronization instructions, loads and stores, are mapped to the SPARC alternate address space instructions with ASI values in the range 0x80 to 0x87 (0x88 to 0x8F is used for uncached synchronization loads and stores). In this way, opcode is actually extended with the ASI value. An extra synchronous trap line is provided in the processor-cache interface for a fine-grain synchronization trap that is induced on a synchronization fault (state miss). The trap line aborts conditional load/store operations and invokes a fault handler that decides what action to take. It can immediately retry the failed operation (spin), or switch to another context and retry the operation later when the given context is active again. For some program-level self-synchronized data structures, such J-structures and L-structures [12], the handler can maintain a centralized queue of threads waiting for a structure element to be filled or to be emptied. Handling a state miss in software, as it is implemented in Sparcle, has the advantage of flexibility though it is expensive. Coarse-grained multithreading supported in hardware is used in Alewife for hiding latency caused by synchronization failures.

We propose to extend the set of FE-memory instructions defined for Sparcle with non-faulting and waiting instructions (see Fig. 3). Non-faulting conditional instructions can be useful for OR-parallelism where, for example, an attempt to write a full write-once variable is not treated as an error and can be ignored. Waiting conditional instructions are provided as a hardware support.

We can distinguish blocking and non-blocking waiting instructions by analogy to conventional blocking and non-blocking loads. If a state miss occurs on a waiting operation, the miss stays in the FE-memory until it is resolved. A waiting instruction might cause the processor to stall whenever a state miss has occurred. In this case, the instruction is blocking. *Non-blocking* waiting loads and stores can be provided that allow an out-of-order processor to continue executing instructions while a state miss is outstanding in the FE-memory system. By analogy to conventional non-blocking loads and stores that prevent stalls on data-cache misses, the non-blocking conditional operations in the FE-memory, can prevent stalls on state misses. This requires a lockup-free FE-cache organization that allows either one (hit-under-state-miss) or multiple (miss-under-miss) outstanding state misses.

4.2.3 Resolving State Misses

A multiprocessor memory hierarchy allows multiple memory requests (cache misses) to be outstanding in the memory system. For achieving waiting FE-memory operations, we propose to keep truck of outstanding state misses in a way similar to keeping truck of outstanding cache misses.

A cache controller already provides some support for holding information about outstanding memory references (cache misses) such as a request table, write buffers, miss information holding registers [8]. A cache controller and a memory controller must also provide buffering for incoming external requests (reads, updates, invalidations, etc.).

A blocking cache and a lockup-free ”hit-under-miss” cache allow only one outstanding cache miss from the processor at a time, while a ”miss-under-miss” lockup-free cache allows multiple outstanding cache misses from the processor at a time.

In a uniprocessor or in a multiprocessor with point-to-point communication, the processor-private cache keeps truck of only its own outstanding references (cache misses). In a multiprocessor with a split transaction bus that provides broadcast ability, a cache controller needs to keep truck of every outstanding miss (request) posted on the bus in order to avoid conflicts and collisions of memory requests.

We propose to treat a state miss as a special case of a cache miss. This should allow maintaining queues of waiters as queues of outstanding cache misses. Design issues to be considers are conflicting misses, merging of misses, bypassing, etc. Ordering of misses from the same processor should be guaranteed automatically. We do not have to provide fairness among multiple processors. In a cache coherence protocol with a full/empty synchronization, we do not need to send data on a read-(read exclusive)-from-full request if the remote location is empty. We believe that the problem with ”allocate on miss” is solvable in this case. This optimization allows to reduce the bandwidth demand on a multiprocessor interconnect (a bus or a network). Finally, we should come up with a cache coherency mechanism that includes full/empty synchronization. Note that, in general, two logical parts of the FE-memory (see Fig. 2), the data memory and the state memory, may use different coherency protocols, for example, a write-invalidate protocol for the data part and a write-update protocol for the state part. With such approach, a coherence mechanism sees two logical caches: one for data and another for the full/empty bits. We intend to evaluate different combinations of different coherency protocols for the FE-state memory.

There are many more design questions to be considered in this project:

1. How should a memory-CPU interface be constructed? What does a processor when a waiting instruction suspends in the memory? One possible solution is to continue with out-of-order execution while it is possible.

2. What to do when the processor finally stalls on instruction(s) that depend on waiting instructions postponed in the memory?

3. What to do when a cache (memory) is saturated with state misses?

4.3 Proposed Evaluation

In order to validate and evaluate the proposed design options, we intend to use a detailed execution-driven simulator such as RSIM [18], (see also http://www.ece.rice.edu/ rsim/). We plan to build a simulation model of a shared memory multiprocessor (for both, SMP or NUMA) with a full/empty memory hierarchy that provides a cache coherency mechanism with embedded fine-grain synchronization. We intend to consider and evaluate the use of different coherency protocols in combination with full/empty synchronization, as well as the use of different coherency protocols for two logical parts of the FE-memory, i.e. the data part and the state part. The functionality of the memory can be verified and evaluated by using parameterized microbenchmark codes.

The performance of the full/empty memory operations can be also evaluated and examined for microbenchmark codes. Workload-driven evaluation of a cache coherency mechanism combined with full/empty synchronization must be done for parallel applications developed (or modified) with use of fine-grain synchronization such as *miccg3d* described in [28].

We intend to use the MIT Alewife machine as a base implementation in our experiments. As the Alewife machine provides special full/empty memory instructions (though a subset of those we introduced and proposed earlier), the binary code obtained on Alewife can be instrumented in order to execute the code on our simulator. Assume, for example, that an application is developed for Alewife with self-synchronized data structures such as J-structures and L-structures, that are accessed with a trapping conditional instructions, such as ”write-and-set-to-full-if-empty-else-trap”. A trap handler that is invoked on a state miss caused by the write to a full J-structure location, maintains (in software) a centralized queue of threads (processes) waiting for synchronization. We could replace the trapping faulting instruction of Alewife with our non-faulting waiting instruction that suspends in the memory on a state miss to be handled in hardware.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, pages 2–13, June 1995.

[2] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An Evolutionary Processor Design for Multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[3] H. Ahmed, L.-E. Torelli, and V. Vlassov. mEDA: A Parallel Programming Environment. In *In Proceedings of the 21st EUROMICRO Conference: Design of Hardware/Software Systems*, pages 253–260, Como, Italy, September 1995. IEEE Computer Society Press.

[4] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *In Proceedings of the International Conference on Supercomputing*, pages 1–6, Amsterdam, The Netherlands, June 1990.

[5] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[6] P. S. Barth, R. S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-strict, Functional Language with State. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, August 1991.

[7] S. H. Bokhari and D. J. Mavriplis. The Tera Multithreaded Architecture and Unstructured Meshes. Technical Report ICASE Interim Report No. 33, NASA/CR-1998-208953, Institute for Computer Applications in Science and Engineering, Mail Stop 403, NASA Langley Research Center Hampton, VA, December 1998.

[8] K. I. Farkas and N. P. Jouppi. Complexity/Performance Tradeoffs with Non-Blocking Loads. In *In Proceedings of The 21st Annual Symposium on Computer Architecture*, pages 211–222, Chicago, Ill USA, April 1994. IEEE Computer Society.

[9] K. Johnson. Semi-C Reference Manual. ALEWIFE Memo No. 20, Laboratory for Computer Science, Massachusetts Institute of Technology, August 1991.

[10] S. W. Keckler, W. J. Dally, D. Maskit, N. P. Carter, A. Chang, and W. S. Lee. Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 306–317, Barcelona Spain, June-July 1998.

[11] A. Kgi, D. Burger, and J. R. Goodman. Efficient Synchronization: Let Them Eat QOLB. In *In Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 170–180, Denver, CO USA, June 1997. IEEE Computer Society.

[12] D. Kranz, B.-H. Lim, A. Agarwal, and D. Yeung. Low-cost Support for Fine-Grain Synchronization in Multiprocessors. In *Multithreaded Computer Architecture: A Summary of the State of the Art*, chapter 7, pages 139–166. Kluwer Academic Publishers, 1994. Also available as MIT Laboratory for Computer Science TM-470, June 1992.

[13] B.-H. Lim and A. Agarwal. Reactive Synchronization Algorithms for Multiprocessors. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 25–35, San Jose, CA, October 1994. ACM.

[14] D. L.Weaver and T. Germond. *SPARC Architecure Manual Version 9, SPARC International*. Prentice-Hall, 1994.

[15] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[16] J. Milewski, L.-E. Thorelli, and H. Wu. Specification of EDA0: An Extended Dataflow Actor Model. Technical Report TRITA-TCS-EDA-9208-R, Dept. of Teleinformatics, Royal Institute of Technology, Stockholm, Sweden, August 1992.

[17] C. A. Moritz and L.-E. Thorelli. A Static Mapping System for Logically Shared Memory Parallel Programs. In *In Proceedings of the 5th EUTROMICRO Workshop for Parallel and Distributed Processing*, London, UK, January 1997. IEEE Computer Society Press.

[18] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *In Proceedings of the Third Workshop on Computer Architecture Education*, February 1997.

[19] B. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *Society of Photo-optical Instrumentation Engineers*, 298:241–248, 1981.

[20] L.-E. Thorelli. The EDA Multiprocessing Model. Technical Report TRITA-IT-R 94:28, Dept. of Teleinformatics, Royal Institute of Technology, Stockholm, Sweden, 1994.

[21] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor. In *In Proc. of the 5th International Symposium on High Performance Computer Architecture*, pages 54–58, January 1999.

[22] V. Vlassov and L.-E. Thorelli. A Synchronizing Shared Memory: Model and Programming Implementation. In *Proc. of the 4th European PVM/MPI User's Group Meeting, (Springer-Verlag Lecture Notes in Computer Science 1332)*, pages 159–166, Novenber 1997.

[23] V. Vlassov and L.-E. Thorelli. Synchronizing Communication Primitives for a Shared Memory Programming Model. In *Proc. of the 4th International Euro-Par Conference, Euro-Par'98 (Springer-Verlag Lecture Notes in Computer Science 1470)*, pages 682–687, Southampton, UK, September 1998.

[24] H. Wu. *Extension of Data-Flow Principles for Multiprocessing*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, Department of Teleinformatics, 1990.

[25] H. Wu, J. Milewski, and L.-E. Thorelli. Sharing Data in an Actor Model. In *In Proceedings of 1992 Int. Conf. on Parallel and Distributed Systems*, pages 245–250, Taiwan, 1992.

[26] H. Wu, L.-E. Thorelli, and J. Milewski. A Parallel Programming Model for Distributed Real-Time Computing. In *In Proc. Int. Workshop on Mechatronic Computer Systems for Perception and Action*, pages 301–308, Halmstad, Sweden, 1993.

[27] K. C. Yeager. The Mips R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[28] D. Yeung and A. Agarwal. Experience with Fine-Grain Synchronization in MIMD Machines for Preconditioned Conjugate Gradient. In *Principles and Practice of Parallel Programming, 1993*, pages 187–197, San Diego, CA, May 1993. IEEE. Also as MIT/LCS-TM 479, October 1992.