Available online at www.sciencedirect.com

ScienceDirect

J. Parallel Distrib. Comput. 68 (2008) 165–181

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Synchronization coherence: A transparent hardware mechanism for cache coherence and fine-grained synchronization

Yao Guo[a,1], Vladimir Vlassov[b], Raksit Ashok[c,1], Richard Weiss[d], Csaba Andras Moritz[e,*]

[a]School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China
[b]School of Information and Communication Technology, Royal Institute of Technology, Sweden
[c]Google Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043, USA
[d]The Evergreen State College, Olympia, WA 98505, USA
[e]Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003, USA

Received 20 December 2005; received in revised form 22 June 2007; accepted 23 August 2007
Available online 4 September 2007

Abstract

The quest to improve performance forces designers to explore finer-grained multiprocessor machines. Ever increasing chip densities based on CMOS improvements fuel research in highly parallel chip multiprocessors with 100s of processing elements. With such increasing levels of parallelism, synchronization is set to become a major performance bottleneck and efficient support for synchronization an important design criterion. Previous research has shown that integrating support for fine-grained synchronization can have significant performance benefits compared to traditional coarse-grained synchronization. Not much progress has been made in supporting fine-grained synchronization transparently to processor nodes: a key reason perhaps why wide adoption has not followed.

In this paper, we propose a novel approach called *synchronization coherence* that can provide transparent fine-grained synchronization and caching in a multiprocessor machine and single-chip multiprocessor. Our approach merges fine-grained synchronization mechanisms with traditional cache coherence protocols. It reduces network utilization as well as synchronization related processing overheads while adding minimal hardware complexity as compared to cache coherence mechanisms or previously reported fine-grained synchronization techniques. In addition to its benefit of making synchronization transparent to processor nodes, for the applications studied, it provides up to 23% improvement in performance and up to 24% improvement in energy efficiency with no L2 caches compared to previous fine-grained synchronization techniques. The performance improvement increases up to 38% when simulating with an ideal L2 cache system.

Keywords: Cache coherence; Fine-grained synchronization; Energy efficiency

1. Introduction

Increasing the available parallelism is perhaps the primary way to improve performance in computer systems. Computer architects and compiler writers are therefore continuously motivated to develop new techniques to capture more program parallelism at various granularity levels such as instructions, threads and processes. To maximize parallelism often speculation-based techniques [9,10,12,16,23,24,27,31,33] are applied. At the same time, improvements in technology enable designers to even increase the grain size of computer architectures.

With increasing levels of parallelism due to both finer-grained systems and more efficient techniques to expose and exploit parallelism, synchronization is set to become a major performance bottleneck and efficient support for synchronization an important design goal. We believe that there are two main directions for efficient synchronization support. One popular approach is based on improving the efficiency of traditional coarse-grained synchronization with speculative execution beyond synchronization points. Another suggested alternative is fine-grained synchronization (FGS), such as synchronization at a word level.

[*] Corresponding author.
E-mail address: andras@ecs.umass.edu (C.A. Moritz).

[1] Yao Guo and Raksit Ashok were at University of Massachusetts while completing most of this work.

0743-7315/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2007.08.003

There are several synchronization mechanisms proposed that use speculation at runtime. Recently proposed speculative synchronization techniques include speculative synchronization [21], speculative lock elision (SLE) [25] and transactional lock removal [26]. Speculation often improves performance by reducing the overhead of false dependencies. While the applicability of speculative synchronization is unquestionable, these approaches have their own share of disadvantages. First, they will likely face scalability limitations when used in finer grained machines. Secondly, power consumption in next generation deep sub-micron technology nodes would likely limit their usefulness further. Simply, speculative execution requires rather complex hardware and could cause significant waste of energy due to unnecessary computations on misspeculations. Nevertheless, it is very clear that speculative synchronization is a great approach as it requires lower programming effort, even if it might trade performance and energy efficiency for it.

Another way to reduce the performance impact of synchronization is by making it more fine grained. Previous research has shown that integrating support for fine-grained (e.g., word level) synchronization can have significant performance benefits compared to traditional coarse-grained techniques [38]. Not much progress has been made, however, in supporting such synchronization mechanism transparently to processor nodes. This is perhaps a key reason why wide adoption has not followed.

This paper explores the idea of *synchronization coherence* (SyC [2]), a transparent fine-grained mechanism using full/empty synchronization [2,17,30], that combines synchronization and caching into one efficient hardware solution. In particular, we propose to handle a full/empty synchronization miss, which occurs when a required full/empty state is not met, in a similar way to a cache miss: the synchronization miss stays in the memory until it is resolved. A cache miss occurs when a target location cannot be read or written in the cache; a synchronization miss occurs when the target location cannot be read or written in the memory. The major difference between a cache and a synchronization miss in SyC is that the former will be eventually resolved whereas the latter can stay in the memory for an arbitrary amount of time. To avoid possible saturation of the memory with synchronization misses, the amount of allowed outstanding synchronization misses can be limited.

SyC has the following key advantages:

(i) It can improve the performance of previous approaches based on full/empty bits for synchronization, due to fewer network messages (or bus transactions) in the SyC protocol and no need to have software trap when synchronization fails.

(ii) It requires minimal changes to cache coherence, because the hardware required by the lockup-free cache organization, which enables outstanding cache misses, can be used for synchronization misses as well.

(iii) It is transparent to processor nodes, because synchronization misses are treated as cache misses and are resolved transparently. An out-of-order processor and a lockup-free miss-under-miss cache organization can hide part of the synchronization miss latency. If a processor cannot continue execution due to a synchronization miss, it stalls or makes a context switch by analogy to a context switch on a cache miss in a distributed shared memory multiprocessor [13].

(iv) It requires likely less hardware overhead compared to speculative synchronization approaches. There is no need for speculation if FGS is properly used.

(v) It is more power efficient than trap-based fine-grained approaches [17,19]. In the case of trap-based approaches to FGS, an interrupt handler either polls the location until synchronization is satisfied, or makes a context switch to another ready thread (if any) after a certain waiting period. Polling wastes CPU time and energy. SyC does not require polling for synchronization because a synchronization miss is treated as a cache miss and it is resolved transparently. In addition, SyC requires fewer network messages that can save considerable energy. Overall, the approach might prove to be also more energy efficient than a speculative execution approach. We do not have an exact comparison due the difficulty of implementing the corresponding speculative schemes.

In order to evaluate SyC, we have developed a complete simulation and compilation flow. We have modified and extended extensively the SimpleScalar simulator [7] to model a directory-based cc-NUMA architecture with full/empty tagged shared memory, cache coherency, and SyC. The simulator implements k-ary n-cube networks and wormhole routing protocols. It is our plan to make this tool available to facilitate additional research on SyC and FGS techniques. In our experiments, we have adapted two applications MICCG3D [38] and LU from the SPLASH-2 suite [35,36] and have developed a new application resembling the communication in DNA chain comparison (also called as Diamond DAG [22]). We also evaluated MST from the Olden benchmark suite [8]. Overall, SyC achieves up to 21% performance improvement and up to 24% power-efficiency benefits compared to state-of-the-art techniques with no L2 caches. With an ideal L2 cache system, we estimate that the performance speedup of SyC over trap-based fine-grained scheme goes up to 38%.

The rest of this paper is organized as follows: Section 2 gives a primer on synchronization including the fine-grained full/empty synchronization semantics as well as architectural support for FGS. Section 3 introduces the proposed SyC approach, including semantics of full/empty memory operations, the proposed architecture and protocol for SyC. Section 4 shows the experimental setup and introduces the applications that we evaluated. Section 5 presents the experimental results. We conclude the paper in Section 6.

2. Overview of synchronization

There are two main types of synchronization: mutual exclusion and condition synchronization. Mutual exclusion guarantees that critical sections of code are not executed by more than

[2] We call it SyC to avoid confusion with SC: sequential consistency.

one thread at a time, whereas condition synchronization delays a thread until a certain condition is true. Locks or semaphores are typically used to control mutual exclusion. Flags, barriers, semaphores, or condition variables with locks or monitors can implement conditional synchronization.

Synchronization can be either *coarse grained* or *fine grained*. The granularity of synchronization is measured by the amount of data that is communicated with the synchronization [17]. For example, barriers are typically coarse grained because multiple shared variables are passed across barriers. Locks, flags and semaphores can be both fine grained and coarse grained.

Coarse-grained synchronization such as a global barrier or a coarse-grained lock obviously can expose false dependencies leading to performance degradation even though the use of coarse-grained synchronization simplifies parallel programming. It is therefore likely less efficient on highly parallel systems. Nevertheless, many approaches, such as improving the performance of coarse-grained locks and barriers [14,37] have been proposed to increase the efficiency of coarse-grained synchronization. We also use an efficient tree-based barrier [37] in our implementation of coarse-grained synchronization.

One approach to reduce false dependencies caused by coarse-grained synchronization, or/and to hide synchronization latency, is to speculatively execute threads beyond synchronization points such as barriers or locks. The recent proposal, speculative synchronization [21], utilizes speculative threads that execute past active barriers. The hardware checks for conflicting accesses and rolls back the offending threads. SLE [25] also reduces the false sharing introduced by sub-optimally placed locks and barriers. Transactional lock removal [26] removes locks to construct an optimistic transaction. In addition, it uses a timestamp-based conflict resolution scheme to resolve data conflicts efficiently.

A finer granularity of synchronization (e.g., at the level of words) is another feasible way to reduce or to avoid false dependencies due to unnecessary synchronization [17,28]. This means, for example, that a thread can wait only for the data item it requires rather than for all shared data passed with a coarse-grained synchronization. FGS allows a dataflow style of computation that suits thread-level parallelism; therefore, FGS could become more important in order to support thread-level parallelism more efficiently in future highly parallel machines and single-chip designs. Traditional synchronization mechanisms such as locks, condition variables, semaphores and barriers can in fact be used for FGS. However, there is a trade-off between granularity of synchronization and the amount of memory used for synchronization: the finer granularity of synchronization, the more memory is required to control it with traditional techniques.

2.1. Full/empty FGS

Another approach to achieve FGS is to implement self-synchronized shared data structures with full/empty state such as write-once I-structures [3], M-structures [4], J-structures and L-structures [1,17]. An instance of a self-synchronized structure can hold a value and can be either full or empty. A structure is accessed with special synchronized reads and writes that can atomically test and change its full/empty state in addition to reading and writing its value.

Architectural support for FGS based on full/empty state includes a full/empty-tagged memory, where each location (e.g., word) can be tagged as full or empty with a full/empty bit associated with it; if the bit is set the location is full, otherwise it is empty. One full/empty bit per a 32-bit word implies an overhead of only 3%, or 1.5% per a 64-bit word. Special loads and stores that can test and/or change the full/empty bit, in addition to reading or writing, are used to access the full/empty-tagged memory. The MIT's Alewife [1] machine, HEP [30], and Tera [2], are examples of multiprocessors with hardware support for FGS using full/empty tags in memory (HEP and Tera use also registers with full/empty bits for synchronization).

For example, MIT's Alewife is a directory-based cc-NUMA multiprocessor with a full/empty tagged shared memory. Hardware support for fine-grained word-level synchronization includes full/empty bits, special memory operations and a special full/empty trap. Software support includes trap handler routines for context switching and waiting for synchronization [19]. While the Alewife architecture supports FGS and shows demonstrable benefits over a coarse-grained approach, it still implements synchronization in a software layer above the cache coherence protocol layer. Keeping the two layers separate entails additional communication overhead.

A few other approaches to FGS exist in other multiprocessors, such as the M-machine with full/empty tagged registers [15] and a simultaneous multithreaded (SMT) processor with hardware-based blocking locks described in [32]. Both mechanisms are proposed for efficient FGS of threads within a processor. However, these designs do not provide FGS across multiple processors.

2.2. FGS structures

In order to define the semantics of full/empty memory operations, let us consider first the J-structures and L-structures, also used to express FGS in the programming environment of the MIT Alewife machine [17].

A *J-structure* is an abstract data type that can be used for consumer–producer type of process interaction. A J-structure has the semantics of a write-once variable: it can be written only once, and a read from an empty J-structure suspends until the structure is filled. An instance of the J-structure is initially empty. A read (J-read) from the full J-structure returns its value, whereas the read from the empty J-structure suspends on the structure until it is filled. A write (J-write) to the empty J-structure makes it full and resumes all pending J-reads (if any), whereas an attempt to write the full J-structure is reported as an error. To be reused, a J-structure can be reset to empty. The semantics of the J-structure are depicted in Fig. 1(a).

A lockable *L-structure* is an abstract data type that has three operations: a non-locking peek (L-peek), a locking read (L-read), and an unlocking write (L-write). An L-peek is similar to a J-read: it waits until the structure is full, and then

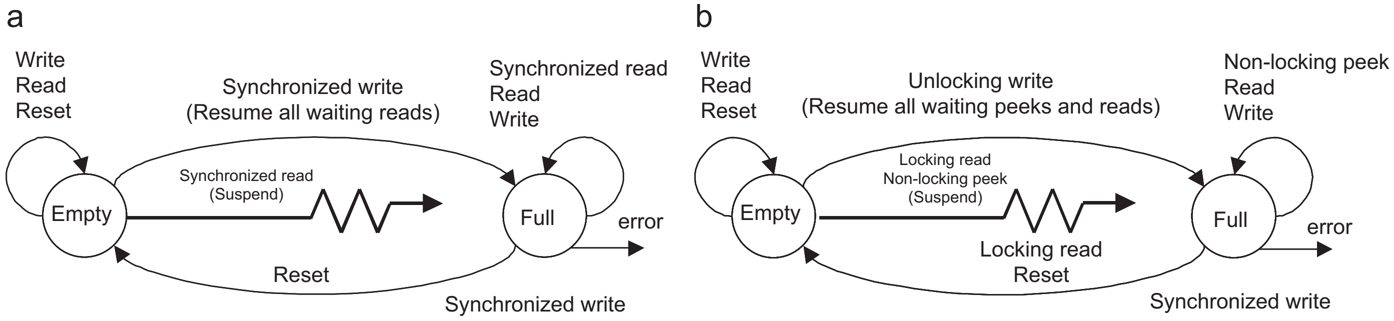

Fig. 1. Semantics of the self-synchronized structures.

returns its value without changing the state. An L-write is similar to a J-write: it stores the value to an empty structure, changes its state to full and releases all pending J-reads, if any. As for J-structures, an error is signaled on a write to the full L-structure. An L-read waits until the structure becomes full then it changes the state to empty (locks the structure), and returns the value read. The semantics of the L-structure are depicted in Fig. 1(b).

3. SyC approach

This section presents SyC, a transparent fine-grained mechanism using full/empty synchronization, that combines synchronization and caching into one efficient hardware solution. We will describe the semantics, architecture and protocol of SyC and compare it with the trap-based fine-grained approaches.

3.1. Semantics of full/empty memory operations in SyC

Assume a multiprocessor with full/empty tagged shared memory (FE-memory) where each location (e.g., word) can be tagged as full or empty (i.e., it has a full/empty bit (FE-bit) associated with it): if the bit is set the location is full, otherwise the location is empty. In order to provide FGS such as J-structures and L-structures described above, the multiprocessor should support special synchronized FE-memory operations (reads and writes) that can depend on the FE-bit and can alter the FE-bit in addition to reading or writing the target location. The processor architecture should include corresponding FE-memory instructions as well as full/empty conditional branch instructions.

We distinguish *unconditional* and *conditional* FE-memory operations. An unconditional operation does not depend on the value of an FE-bit. We assume that a conditional read is executed if the location is full, whereas a conditional write is executed if the location is empty. A *synchronization miss* occurs when the required state of the location is not met. We propose that each conditional operation has two versions: a *trapping* (or faulting) version that traps on a synchronization miss, and a *waiting* version that is postponed on a state miss until the location reaches the required state.

We distinguish *non-altering* and *altering* FE-memory operations. Non-altering operations do not change the FE-state. An altering operation (read or write) sets a new FE-state to the location beyond reading or writing data. We assume that the altering read sets the location to empty, and the altering write sets the location to full.

Finally, we assume that each of the FE-operations returns, as a side effect, an original value of the full/empty bit associated with the location. This is to be used as a full/empty condition code in the processor.

The semantics of the synchronized conditional (waiting and trapping) FE-memory instructions and their altering versions are shown in Table 1. We use angle brackets ⟨and⟩ to denote an atomic action; functions *wait* and *notifyAll* are similar to the wait and notifyAll methods in Java monitors: wait suspends a thread on the location until the thread is notified, whereas notifyAll resumes all threads pending on the location (if any).

Semantics of synchronized waiting FE-memory operations can be implemented using synchronized trapping FE-memory instructions like in the MIT Alewife multiprocessor [17]. On a synchronization miss, a trapping FE-memory instruction fires a trap, and an interrupt handler can either poll the location until the required full/empty state is met, or suspend the thread, place it on a queue of waiters and switch the context to another ready thread (if any). With trapping FE-memory instructions, the queue of waiters, i.e., threads suspended on synchronization misses, is maintained in software. When an FE-bit is altered, the corresponding queue is checked and if it is not empty, all (or selected) waiters can be resumed, i.e., moved to the queue of ready threads.

3.2. Architectural support for SyC

In order to support SyC, some changes are required to the architecture of a typical cc-NUMA multiprocessor. This section describes the architectural enhancements that need to be made.

Each word in the shared memory is tagged with a full/empty bit (*FE-bit*) and a pending bit (*P-bit*). An FE-bit indicates whether a corresponding word is full or empty. A P-bit indicates whether there are operations (reads or writes) pending on a corresponding word: if the P-Bit is set, it means there are pending synchronized reads (if the word is empty) or pending writes (if the word is full) for the corresponding word. This information is required so that a successfully executed altering operation can immediately satisfy one or more pending operations.

Table 1
Conditional full/empty tagged memory instructions

FE-memory instruction	Notation	Semantics
Waiting Read	WRd	⟨while (FE-bit == empty) wait(); read;⟩
Trapping Read	TRd	⟨if (FE-bit == empty) trap else read;⟩
Waiting Write	WWr	⟨while (FE-bit == full) wait(); write;⟩
Trapping Write	TWr	⟨if (FE-bit == full) trap else write;⟩
Waiting Altering Read	WARd	⟨while (FE-bit == empty) wait(); read; set FE-bit to empty; notifyAll();⟩
Trapping Altering Read	TARd	⟨if (FE-bit == empty) trap else {read; set FE-bit to empty; notifyAll();}⟩
Waiting Altering Write	WAWr	⟨while (FE-bit == full) wait(); write; set FE-bit to full; notifyAll();⟩
Trapping Altering Write	TAWr	⟨if (FE-bit == full) trap else {write; set FE-bit to full; notifyAll();}⟩

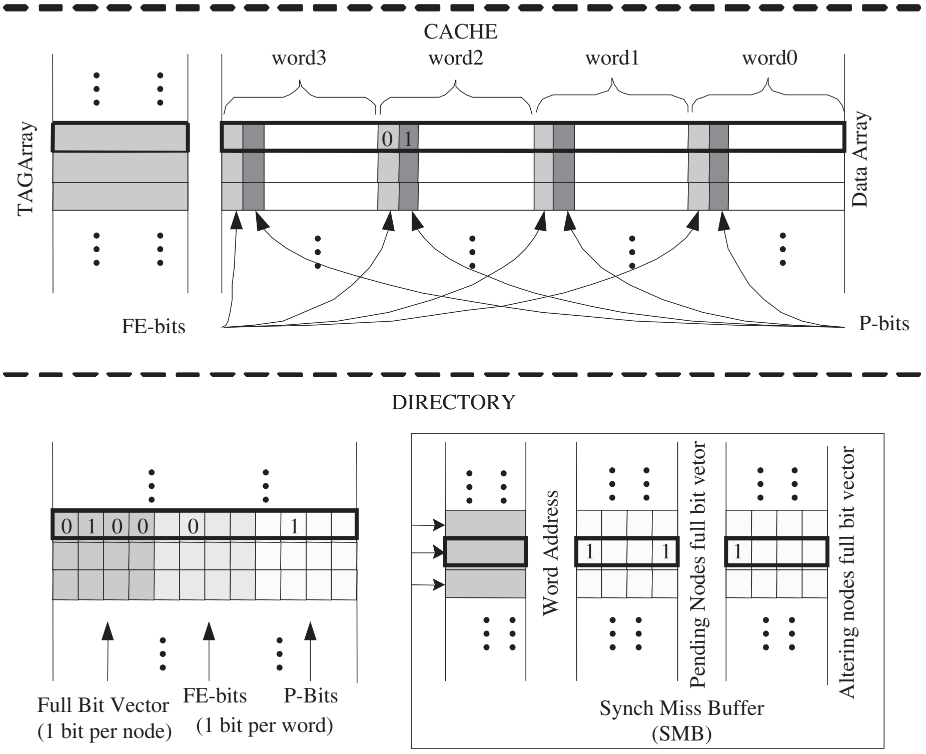

Fig. 2. Organization of the full/empty tagged cache and directory.

A vector of FE-bits and a vector of P-bits associated with words in a memory block are stored in the coherence directory and as an extra field in the cache tag when the block is cached. This way, a tag (directory) lookup includes tag match and inspection of full/empty bits. Each home node (directory) also contains a *state miss buffer* (SMB) that holds information regarding which nodes have pending operations for a given word and whether operations are altering or not. The synchronization miss is treated as a cache miss, and information on the miss is recorded at the cache, e.g., in a Miss Information/Status Holding Register (MSHR) and a write buffer [11].

Fig. 2 illustrates the changes required to cache and directory structures. In the figure, we assume a 4-processor system with a 32-byte (4 words) memory block. In addition to the 256-bit data block and the tag bits, each cache block has 8 extra bits: 4 FE-bits and 4 P-bits (that is about 3% of storage overhead per cache block).

As already mentioned, information regarding which nodes have pending operations for a given word is stored in the SMB, which is indexed by the word address (see Fig. 2). An entry in the SMB is allocated when a first operation suspends on the word; an entry is released when there are no pending operations (the pending queue is empty). Each entry in the SMB contains two full bit vectors: a vector of pending nodes and a vector of altering nodes.

A bit in the full vector of pending nodes corresponds to a node and indicates whether the node has conditional FE-memory operations (reads or writes) pending on the word. A bit in the full vector of altering nodes indicates whether the pending operation is altering. This information is needed to resume (and to complete) pending operations when the FE-bit of the word is altered. If the vectors indicate that there are both types of operations, altering and non-altering, pending on the word, then several (if not all) non-altering but only one altering operations can be resumed. For example, the directory controller can resume all non-altering operations and one altering operation. When the operations are resumed the controller sends corresponding messages to pending nodes indicating which operations have been resumed and alters the FE-bit if one of the resumed operations is altering. The resulting FE-bits are always sent with replies to indicate the current FE-state of a word in the memory block. Similarly to the MIT Alewife multiprocessor [1], the FE-bits can be transmitted with the address.

How many entries should the SMB contain? For a multiprocessor with n nodes and hit-under-miss lockup-free caches, there could be at most $n-1$ pending operations (reads or writes). Hence, for a 4-processor system, for example, 3 entries would be enough in the SMB of each node. For very large configurations or for a multiprocessor with miss-under-miss lockup-free caches (not used here), the required number of entries may become too large. In such cases, an overflow mechanism can be employed: if a directory runs out of SMB entries, the directory controller should not accept the request that caused the overflow and send it back to the cache controller to retry.

In SyC, synchronization misses to the same word form a list of pending FE-memory operations that is maintained in hardware by cache and directory controllers, rather than in software by an interrupt handler as in the previous approaches [1]. Thus, SyC allows implementing conditional waiting FE-memory operations transparently to the CPU.

The SyC protocol calls for slightly more sophisticated directory and cache controllers. The cache controller not only matches the tag but also checks the full/empty bits depending on the instruction, and makes a decision based on the state of the cache line as well as the associated FE-bits and P-bits. The directory controller is also modified to account for the SMB implementing the SyC protocol in the directory. It has to send data asynchronously to resolve synchronization misses on writebacks, by looking up the SMB, etc. More details of the SyC protocol are provided next.

3.3. *SyC protocol*

We have chosen a directory-based cache coherence protocol used in the SGI Origin [18] multiprocessor as the baseline for our SyC protocol. This is the MESI protocol, which employs request-forwarding transactions involving three nodes: a requesting node, a home node and an owner of a recent copy of a target memory block. It allows cache blocks in the *Exclusive* state to be replaced without requiring notification to the directory; it fully supports upgrade requests, etc. The directory side has five states: *Un-owned* (block not present in any node's cache), *Exclusive* (block may be present in only one node's cache), *Shared* (block may be present in several nodes' caches), *Busy-Exclusive* (directory is busy with exclusive-read request forwarding), and *Busy-Shared* (directory is busy with shared-read request forwarding). Further details can be found in [18].

To integrate FGS into MESI, several new messages have been added. SyC supports the complete set of FE-memory operations including eight conditional operations shown in Table 1 and unconditional operations (not shown in the table): ordinary read and write, altering read and write. We do not consider here ordinary reads and writes because they are handled in SyC in the same way as in the SGI Origin protocol.

The states of cache lines and directory states of memory blocks are the same as in the MESI protocol, however, each word in a cache or in the main memory has a synchronization state: a word can be either full or empty that is indicated by the FE-bit, and it may have a queue of pending FE-memory operations that is indicated by the P-bit. A home directory controller using the FE- and P-bits and SMB at the directory side maintains a queue of pending conditional (waiting) operations.

To illustrate the inner workings of SyC, we consider several examples of FE-memory operations. The examples are shown in Fig. 3. Namely, we consider operations (waiting read and altering write ones) needed to support J-structures. Suppose a processor issues a synchronized conditional read that misses in its cache (it could be either ordinary miss or synchronization miss).

Fig. 3(a) shows a scenario on a synchronized waiting read. The cache sends a RD_SYNC message (shown as #1 in the figure) to the directory, which, assumed here, has the required word in the full state (FE-bit is 1). The directory therefore sends a SHD_REPLY (if its in the Shared state) or an EXCL_REPLY (if its in the unowned state) and the synchronized read is satisfied once the message reaches the cache (#2).

In Fig. 3(b), the cache again sends a RD_SYNC message to the directory (#1). This time, however, the word is empty (FE-bit is 0) and there are no synchronized reads already pending (P-bit is 0). If the directory state is unowned or shared, there cannot be a node that has done a synchronized altering write to its local cache. In this case, the directory sets the P-bit (#2) and creates an entry in the SMB with the requesting node's bit set (#3). When a synchronized altering write occurs, the SMB will be checked and replies sent, to satisfy all the pending synchronized reads to this word, as will be seen later.

Fig. 3(c) shows a situation similar to above, except that the memory block is in the Exclusive state at the directory. There is a chance therefore, that the owner of the block might have done a synchronized write to this word. The directory therefore forwards the request by sending an INTERVENTION_SYNC request (#4) to the owner and passes to the Busy state. The owner cache, however, does not have the requisite word in full state (FE-bit is 0) either, and therefore sets the P-bit (#5) so that if a synchronized write is performed later on, the pending reads can be immediately notified. It also sends a SYNC_NAK message and the directory, upon receiving this message, passes from the Busy state back to the Exclusive state (#6).

Fig. 3(d) shows the sequence for a synchronized (trapping) altering write. The FE-bit is 0 (an exception if it was already 1) and there are pending requests (P-bit is 1). The cache sets the FE-bit and passes to the Shared state (#2). It then sends a synchronized writeback message (SYNC_WB) to the directory. The directory on getting this message (#3) sets the FE-bit, resets the P-bit and passes to the Shared state. The SMB entry is checked (#4) to find out the pending nodes. Assume that there

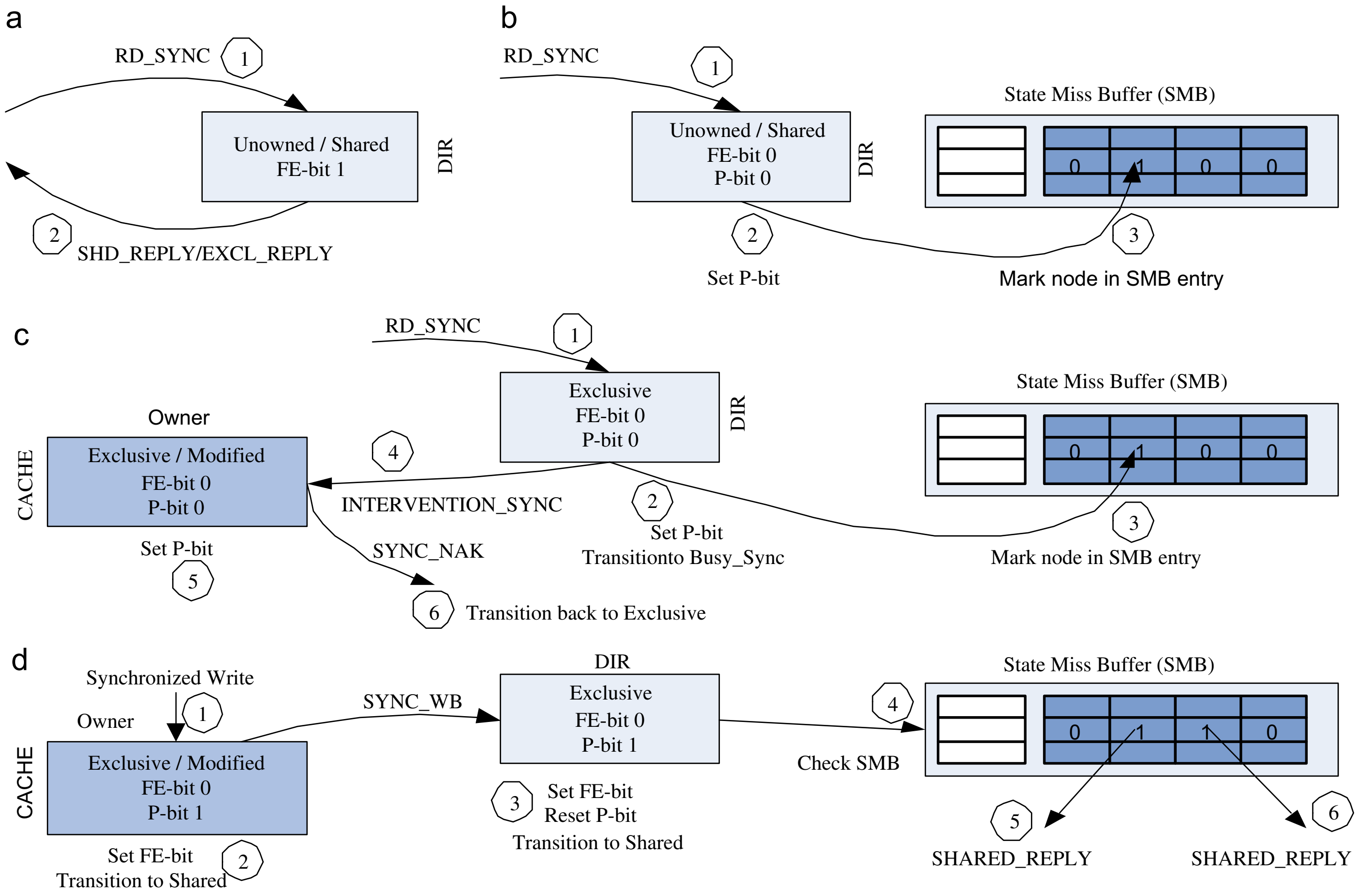

Fig. 3. Examples of SyC protocol transactions.

are no pending altering reads. SHARED_REPLY messages are sent to each of these pending nodes (#5 and #6) to satisfy the synchronized reads, and the SMB entry is then squashed.

The new messages added to the coherence protocol to integrate FGS (waiting and altering FE-memory operations) are RD_SYNC, WR_SYNC, INTERVENTION_SYNC, SYNC_NAK, and SYNC_WB and the directory has the additional state BUSY_SYNC. Each RD_SYNC or WR_SYNC request also indicates whether the corresponding operation (read or write) is altering. Many of the protocol messages also require the FE-bits and P-bits to be tagged along.

To support all FE-memory operations, the protocol distinguishes altering and non-altering operations: an altering operation alters the FE-bit, whereas a non-altering read does not change the FE-bit. If an altering operation causes some pending orthogonal altering operations to be resumed (for example, an altering write resumes a waiting altering read) then the SyC protocol uses a “passing the baton” technique so that the FE-bit is not changed by either of the operations.

To illustrate this, consider the following example. Suppose a processor issues a conditional altering write to a word, which misses in the cache. In this case, the write miss is recorded in the MSHR and is forwarded to the home node. Assume that the directory state of the target memory block is shared or unowned, and the target word is empty (the FE-bit is 0). The directory controller checks the P-bit, and if it is not set (there are no waiters) the directory controller sets the FE-bit to full and processes the synchronized write miss as an ordinary write miss. If the P-bit is set, there are pending reads on the word, and the directory controller looks up the SMB to find which nodes have conditional reads pending for this word and whether some of the pending reads are altering. The controller picks all nodes with non-altering reads and one node with altering read and resets corresponding bits in the SMB and the P-bit in the directory. It sends the data for the block back to the requestor with a forwarded request to resume selected pending nodes. The requester writes the block and replies directly to the pending nodes, sending a revision message to the home node. Note that the directory state is shared and the word is left empty.

4. Experimental setup

In order to evaluate SyC, we have modified and extended extensively the SimpleScalar simulator [7]. The basic structure of the simulator, including the synchronization primitives, follows the conventions used by the multiprocessor version of SimpleScalar developed by Manjikian [20], however, our simulator is based on the out-of-order version of SimpleScalar

Table 2
Multiprocessor parameters

L1 D-Cache	32 kB, 4-way, 32 byte per line
L1 latency	1 cycle
DRAM latency	100 cycles
Interconnect layout	2-D mesh and Hypercube
Flit size	32 bits
Interconnect speed	4 cycles per hop
Router delay	4 cycles for the first flit message
Launch delay	4 cycles
Trap cost (for baseline)	10 cycles
Number of processor nodes	1, 4, 8, 16, 32, 64

(sim-outorder) with detailed timing simulations. We model a directory-based cc-NUMA architecture with full/empty tagged shared memory and support for SyC. In the simulated network, messages are broken into flits and sent in a pipelined manner. Table 2 shows some of the important simulation parameters.

We model two types of interconnection networks: 2-D meshes and hypercubes. Time (hops) needed for communication between two nodes in the network are calculated based on the interconnect types for each of them. For a 2-D mesh network, the interconnect delay (from 1 to $(X + Y - 2)$, where X and Y represent the width and height of the mesh and $X * Y = N$ is the number of nodes) is dependent on the individual position of the two communicating nodes, while the delay is typically smaller ($\leqslant \log_2 N$, where N is the number of nodes) in the case of hypercubes. Network contention is also considered during the calculation.

To express FGS, we employ L-structures and J-structures similarly to the MIT Alewife machine [17]. To access the structures, we have developed synchronized load and store functions using FE-memory operations. We have extended the compilation flow to automate support for source level FGS. The FE-instructions that implement FGS are inlined during compilation as assembly macros.

4.1. Applications

During our research we have noted that there are very few benchmarks available that have been written for evaluating FGS; previous studies [38] have primarily focused on MICCG3D. The major reason for this is the lack of machines that support FGS in hardware as well as the lack of compiler support. Most of the shared memory applications are written using traditional software synchronization mechanisms such as barriers, semaphores, locks and condition variables. FGS often would require changing the underlying algorithms used.

We have used three available applications, MICCG3D [38], LU from the SPLASH-2 suite [35,36] and MST from the Olden benchmark suite [8]. We also developed a new application resembling the communication in DNA chain comparison (also called as Diamond DAG [22]).

Other benchmarks in SPLASH-2 have also been examined, but we found that their data access patterns do not benefit from FGS without perhaps a complete redesign of the algorithms and modification of the underlying data structure. Based on our analysis, most applications in SPLASH-2 have limited number of barriers and thus the synchronization overhead is already pretty small. According to the original SPLASH-2 paper [36], most of the applications have a close to perfect speedup when a perfect memory system is applied, with the exception of only four benchmarks: *LU*, *cholesky*, *radix*, and *radiosity*. We consider therefore these benchmarks as possible targets for FGS and studied them in detail. A summary of our analysis is shown below.

- *LU*: The LU kernel factors a dense matrix into the product of a lower triangular and an upper triangular matrix. FGS has proved to be beneficial for LU during our experiments and has been used in our evaluation.
- *Cholesky*: The blocked sparse Cholesky factorization kernel is similar in structure and partitioning to the LU factorization kernel but has two major differences: (i) it operates on sparse matrices, and (ii) it is not globally synchronized between steps. Without frequent global synchronization (as in LU), cholesky does not benefit significantly from FGS.
- *Radix*: The integer radix sort kernel implements a parallel sorting algorithm based on counting sort. Since part of the algorithm during the prefix computation in each phase is not inherently parallelizable, even FGS cannot make it more efficient algorithmically.
- *Radiosity*: This application computes the equilibrium distribution of light in a scene using the iterative hierarchical diffuse radiosity method. The lower speedup of radiosity is due to its scalability problem caused by its small problem size. We cannot unfortunately generate a larger input size. This is because these inputs have a special meaning, so using random sets would not work.

Therefore, only LU from SPLASH-2 suite is used in our evaluation. We have developed two versions of each application: a version with coarse-grained barrier synchronization (henceforth called *coarse-grained version*), and a version with FGS using J-structures (*fine-grained version*). Following is a brief introduction to the applications.

4.1.1. DNA chain comparison

Fig. 4 shows the data flow pattern for the DNA chain comparison application. Each node in a thread represents a computation phase. The arrow shows the data dependency between phases: computation in each thread on each phase depends on the values previously computed by its two adjacent threads and itself. Such a dataflow pattern requires lots of synchronization and can therefore likely benefit from a FGS approach.

4.1.2. MICCG3D

MICCG3D is a Modified Incomplete Cholesky Conjugate Gradient method for 3-D boundary value problems. It solves the

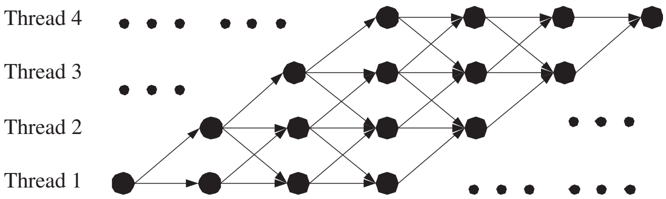

Fig. 4. Data flow pattern for the DNA chain application.

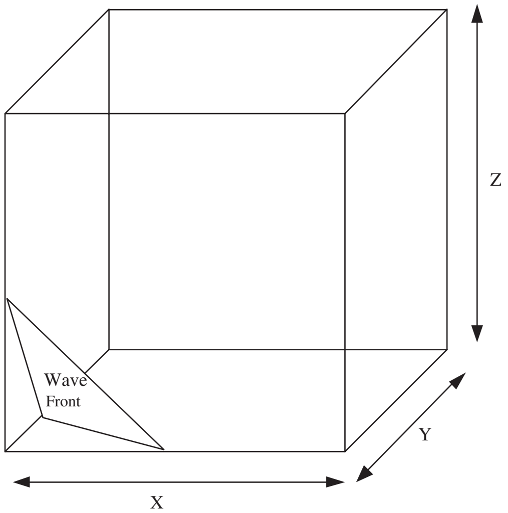

Fig. 5. MICCG3D computational wavefront [38].

matrix equation: $Ax = b$, where A is a sparse and symmetric positive matrix, x is the vector that needs to be solved. The computation of MICCG3D is illustrated in Fig. 5 [38]. Due to the inherent dependency within x, the data being computed at an instant of time forms a "wave front". All the data on the same wave front can be computed in parallel and the dependency is perpendicular to the front.

We have implemented both fine-grained and coarse-grained versions of this application. In the coarse-grained implementation of MICCG3D, the data are partitioned into blocks. Barriers are inserted to maintain the dependency across threads. In the fine-grained implementation, we use J-structures supported by FE-memory operations to enforce the dependency among threads.

A detailed description of MICCG3D can be found in [38].

4.1.3. LU

LU is an algorithm to factorize a dense matrix that can be performed efficiently if the dense $n \times n$ matrix A is divided into an $N \times N$ array of $B \times B$ blocks ($n = NB$). Fig. 6 shows the pictorial representation of the algorithm [36]. The arrows in the figure show the dependencies among blocks. We used the original implementation for LU found in [36] as the coarse-grained version of the application. We converted it to a fine-grained one by eliminating barriers and adding FE-memory operations in the source code. We have not changed the way data is partitioned. In our future work, we plan to completely redesign these applications to better suit FGS.

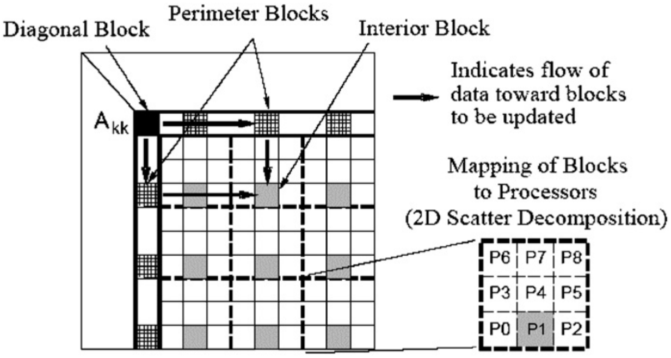
Diagonal Block
Perimeter Blocks
Interior Block
A_{kk}
Indicates flow of data toward blocks to be updated
Mapping of Blocks to Processors (2D Scatter Decomposition)
P6 P7 P8
P3 P4 P5
P0 P1 P2

Fig. 6. LU factorization of blocked dense matrix [36].

4.1.4. MST

MST is a parallel algorithm calculating minimum spanning trees based on a given non-directed weighted graph included in the Olden benchmark suite [8]. The application implements the parallel algorithm proposed in [5]. It is also studied in the speculative synchronization [21]. The Olden codes are pointer-based applications that operate on graphs and trees. They are annotated so that the compiler or the programmer can easily parallelize them. The only difference for MST is that we run it on up to 32 processors (instead of 64 for other applications) based on the suggestions in the benchmark distribution.’

4.2. Programming support for SyC

Appropriate programming support is very important in order to make FGS implementation more effectively. Here we describe the essential programming support required at both application-level and compiler-level for FGS.

Application-level support helps programmers to write parallel programs with FGS more efficiently. Specifically, the required elements could include:

- First, one must understand the implications at the algorithm level in order to exploit the inherent benefits of FGS.
- Second, a mechanism to express the variables as FGS-based structures should be provided, such as using the L- or J-structures. The fine-grained structures should be declared as language provided special types (for example, new modifiers in C/C + + could be used).
- Fine-grained structures can also be used in conjunction with higher-level synchronization constructs such as semaphores and monitors. These structures can be based on FGS. For example, efficient FGS-based lock and barrier implementations such as optimistic synchronization [28] primitives using load linked/store conditional would be an attractive alternative to traditional mutual exclusion locks. As reported by Martin Rinard, the use of FGS-based synchronization primitives can significantly reduce the memory consumption and improve the overall performance [28].

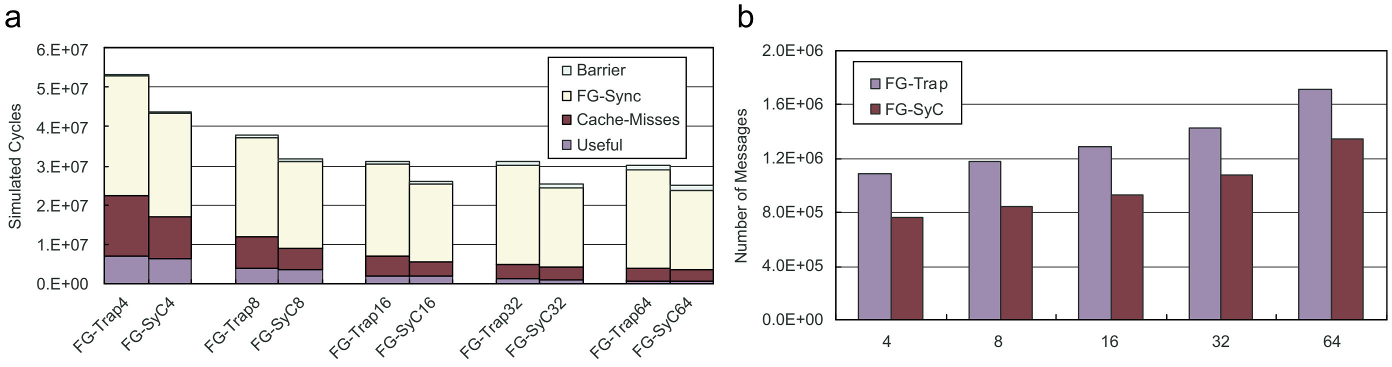

Fig. 7. (a) Execution time and (b) number of messages of application DNA chain under baseline (FG-Trap) and synchronization coherence (FG-SyC).

At the compiler-level, any memory access that goes to FGS data structures defined above has to be translated into special load/store operations, which will be recognized by the architecture.

We assume that the application is written explicitly in parallel in the above discussions. If the original application is written in sequential format, parallelizing compilers would be needed to: (1) identify parallel components in the application; and (2) identify the structures where FGS could be used and transform them to special FGS load/store operations automatically. Apparently, this will be much more difficult compared to the previous approach. Nevertheless, single-chip multiprocessors (especially) would benefit to have such compilers. The techniques required would be a relatively small extension to compilers such as the one designed for Raw processors [34].

5. Results

In order to estimate performance improvements and energy savings due to SyC, we have conducted three series of simulations: coarse-grained barrier synchronization on a traditional directory-based cc-NUMA architecture (Coarse), FGS on a directory-based cc-NUMA architecture with full/empty tagged shared memory (FG-Trap) and the proposed SyC (FG-SyC).

As expected, both fine-grained versions (FG-SyC and FG-Trap) achieve good performance speedup compared to Coarse. FG-SyC also achieves significant speedup compared to FG-Trap in most cases. In addition, FG-SyC achieves significant energy savings compared to both FG-Trap and Coarse. In cases where FG-SyC does not show clear performance benefit compared to FG-Trap, it still saves overall energy significantly. In contrast, FG-Trap is a little bit worse in energy compared to Coarse in some cases. Overall, FG-SyC is consistently better than FG-Trap when both performance and energy are considered. Detailed results and analysis will be presented next.

We first show the performance and energy consumption comparison with fixed data size and 2-D mesh interconnect network. Then, we compare the speedup with different network configurations (2-D mesh and hypercube) and different data sizes; we also estimate the impact of L2 caches.

5.1. Performance improvement

Fig. 7(a) shows the breakdown of execution time for the DNA chain application. “FG-Trap” corresponds to the baseline, where FGS is implemented on top of cache coherence, like in the MIT Alewife [1]. “FG-SyC” refers to our new approach, SyC. From the result we can see that FG-SyC outperforms FG-Trap significantly, ranging from 19% to 23% with an average speedup of 21%.

The execution time is further broken down into several categories. “Useful” is the time spent on computation. “Cache-misses”, “FG-Sync” and “Barrier” are different sources of overhead. “Cache-misses” cycles are CPU stalling cycles caused by regular cache-misses, including both local cache misses and remote cache misses in the shared memory multiprocessor system. “Cache-Misses” are determined by local memory access latency, network configuration, traffic and the remote memory access latency. “FG-Sync” represents synchronization overhead for fine-grained synchronization: either the cycles spent on a trap to handle FGS misses for FG-Trap, or the waiting cycles due to synchronization misses for FG-SyC. Note that for FG-SyC, there is no need to trap on synchronization misses, since they are handled transparently. “Barrier” is the number of cycles spent primarily on barriers.

From the above results we can see that the performance improvement primarily comes from the reduced “Cache-misses” cycles. This is because FG-SyC generates fewer messages and reduces the network traffic (as shown in Fig. 7(b), FG-SyC reduces the number of messages ranging from 21% to 30%, with an average of 26%). In addition, we can see that the “FG-Sync” in FG-SyC is smaller than its counterpart in FG-Trap. This is mainly due to the fact that FG-SyC handles FGS more efficiently by replacing software traps with (synchronization) cache misses.

Fig. 8(a) shows the execution time breakdown for MICCG3D. We evaluated both coarse-grained and fine-grained versions. The data size is 8 × 8 × 256. In the figure, “Coarse” represents the coarse-grained implementation.

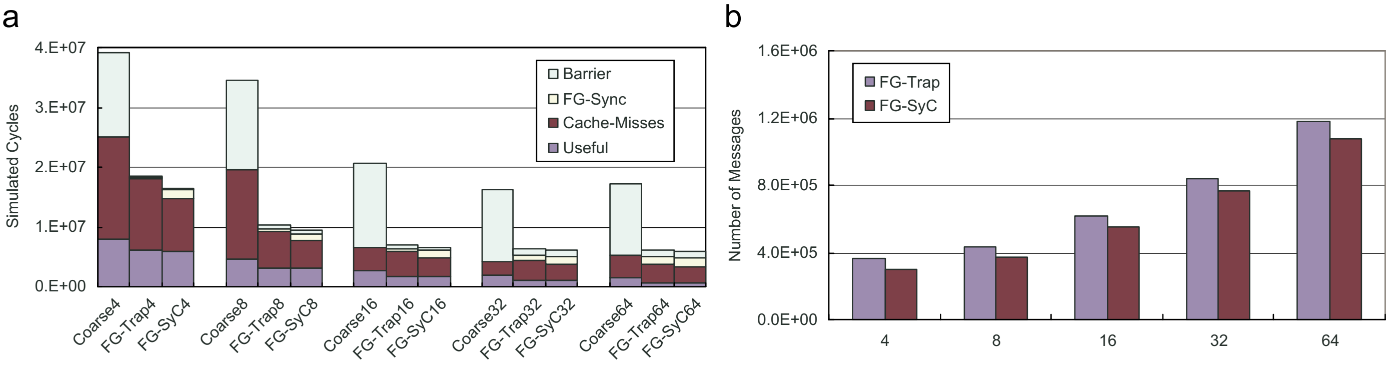

Fig. 8. (a) Execution time and (b) number of messages of application MICCG3D under baseline (FG-Trap) and synchronization coherence (FG-SyC).

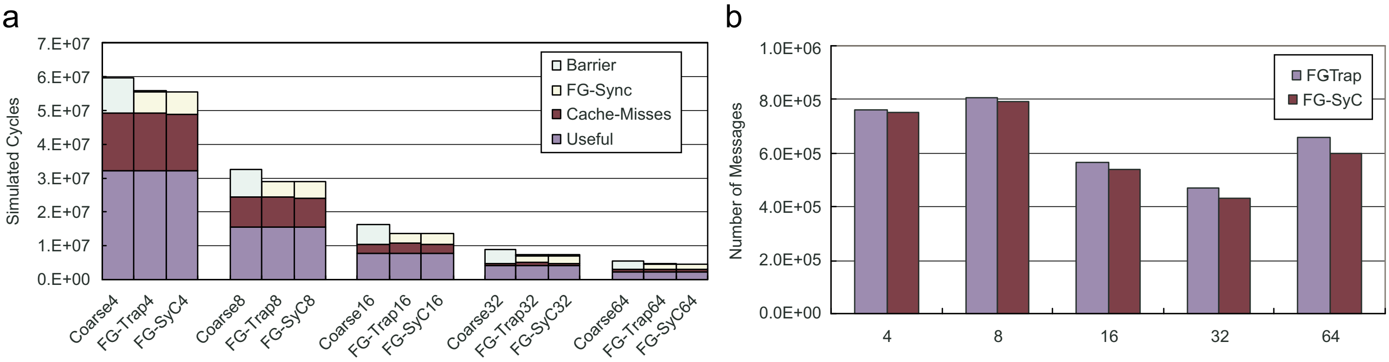

Fig. 9. (a) Execution time and (b) number of messages of application LU under baseline (FG-Trap) and synchronization coherence (FG-SyC).

We can see clearly from the above graphs that FGS has significant advantage over the coarse-grained version. FG-Trap outperforms coarse-grained by a factor of 2.4–3.7X. The numbers are fairly close to those in [38]. FG-SyC shows additional improvement over FG-Trap. The speedup over FG-Trap ranges from 3% to 12%, with an average of 6.7%.

From the execution time breakdown, we can see that FGS (both FG-Trap and FG-SyC) reduces the "Barrier" cycles significantly, which is the main source of the performance improvement. FG-SyC also improves the performance over FG-Trap by reducing "Cache-misses" cycles (reflected by the reduced number of network messages, as shown in Fig. 8(b), because of the same reason as in the DNA chain application).

Besides the speedup achieved, FG-SyC also reduces the number of messages (shown in Fig. 8(b)) by 8–16%, with an average of 11%.

Fig. 9(a) shows the performance of LU, on both coarse-grained and fine-grained versions. The input data size is 256 × 256. In this application, we do not observe much speedup of FG-SyC over the FG-Trap baseline. As seen in the graph, the "Barrier" cycles in the coarse-grained version are mostly converted to "FG-Sync" for both fine-grained schemes. Also, we notice that the "FG-Sync" cycles for FG-Trap and FG-SyC are almost equal. This means that for this application, those synchronization overheads are inherent to the application and we cannot do much about it.

However, FG-SyC can still reduce the number of network messages (see Fig. 9(b)), which helps reducing the total power consumption, as we will show in the next section. The speedup of FG-SyC over Coarse ranges from 7% to 24%, with an average of 17%. In Section 5.3.3, we show that this speedup could be further improved with L2 caches.

Fig. 10(a) shows the execution time for MST on both coarse-grained and fine-grained versions. As mentioned previously, we simulated the application with up to 32 processors based on the suggestions from the benchmark developer. The speedup of FG-SyC is ranging from 14–41% over Coarse and 7–15% over the FG-Trap baseline. FG-SyC can also reduce the number of network messages (see Fig. 10(b)) for up to 7% over FG-Trap (except for the 4-processor case, where we observe a slight increase in network messages).

5.2. Energy consumption

We modified the Wattch simulator [6] to calculate the energy consumption in a multiprocessor. Energy cost for each processor is calculated individually and then summed up. Energy cost for the extra hardware such as FE-bits and P-bits in caches are accounted for in the energy simulation with the Cacti [29] model.

To estimate the off-chip message energy cost (in CPU), we assume that each message costs 64X or 256X of a single word L1 cache access, and present results for both cases. Similar assumptions on off-chip memory energy cost have been made in [39]. The actual energy cost for off-chip messages depends on the number of bits in transition on IO pads and specific

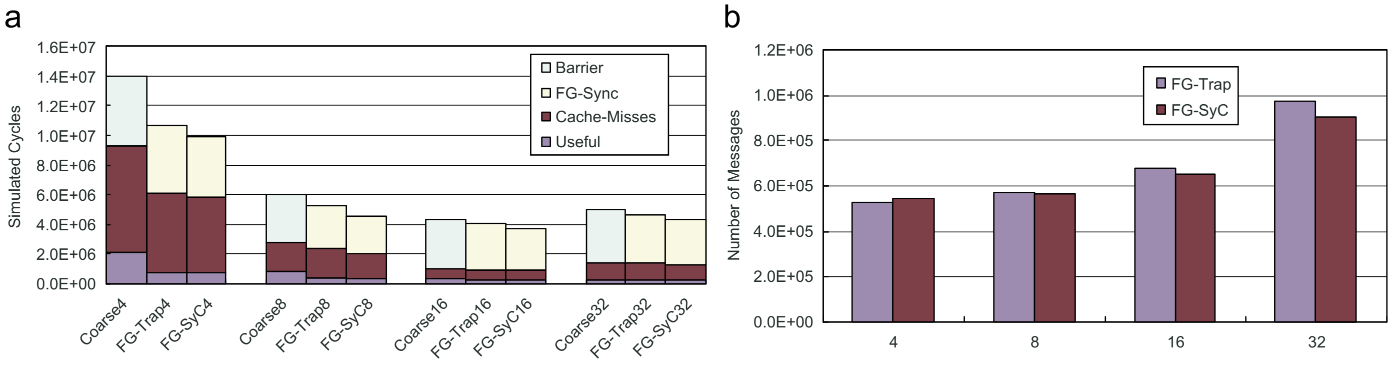

Fig. 10. (a) Execution time and (b) number of messages of application MST under baseline (FG-Trap) and synchronization coherence (FG-SyC).

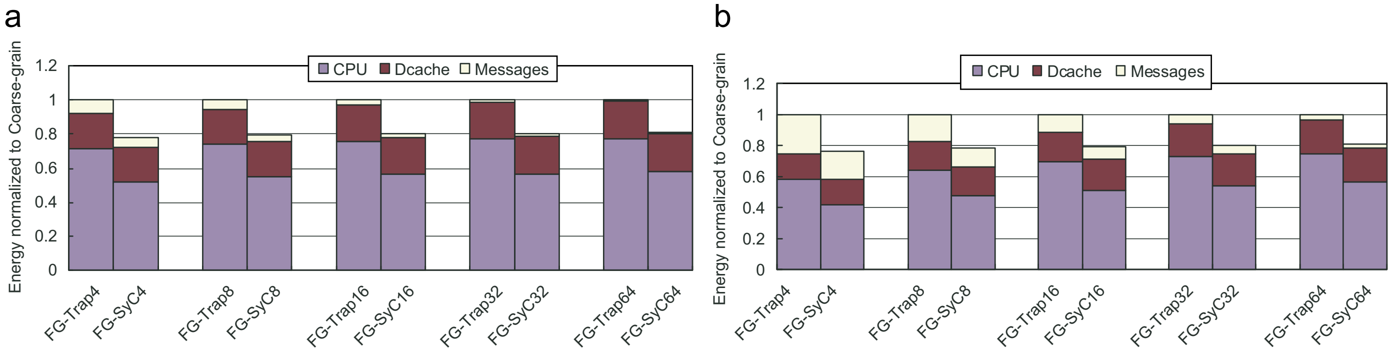

Fig. 11. Energy consumption for the DNA chain application: (a) message cost as 64X of L1 cache energy cost; (b) message cost as 256X of L1 cache energy cost.

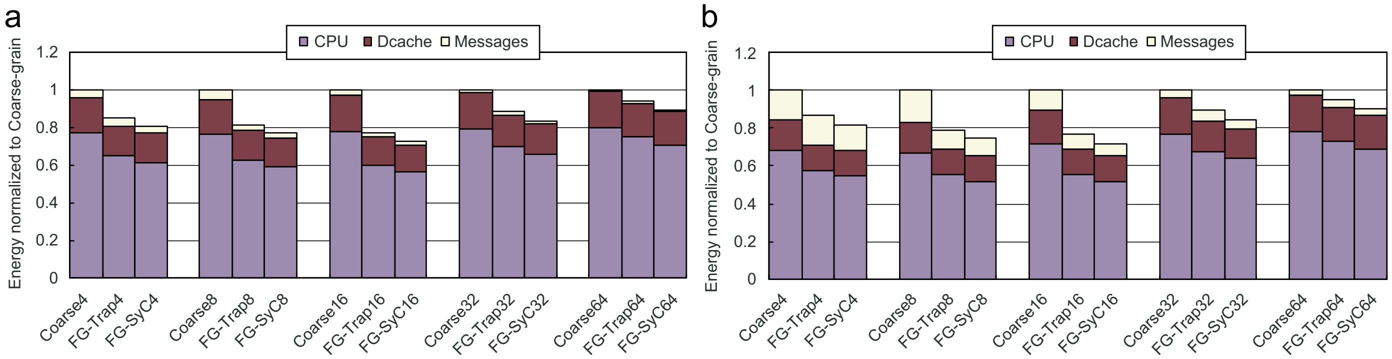

Fig. 12. Energy consumption for MICCG3D: (a) message cost as 64X of L1 cache energy cost; (b) message cost as 256X of L1 cache energy cost.

packaging/implementation details. The 64X case is more representative of current systems, while the 256X is a more pessimistic projection for future generation systems.

The energy results for the DNA chain application are shown in Fig. 11. We show breakdown numbers on energy consumptions for CPU, D-cache and messages, respectively. All the energy numbers are collected from Wattch and then normalized to the FG-Trap total energy cost. The figure shows that we save a range of 18–22% chip-wide energy across all processor nodes for the 64X case. This is due to the elimination of busy waiting and less network traffic generated by FG-SyC compared to FG-Trap. The energy savings in CPU are due to the fact that FG-SyC transformed the busy waiting during polling in FG-Trap into synchronization (cache) misses. While polling wastes significant energy, the synchronization misses in FG-SyC do not execute instructions thus being more energy efficient. When the message energy cost increases to 256X (future systems), the savings become larger, up to 19–24%, because FG-SyC also reduces the number of messages significantly compared to FG-Trap.

Fig. 12 shows the energy results for the application MICCG3D, which is normalized to the total energy cost of the coarse-gained case. The figure shows that both fine-grained cases are much more energy efficient than the coarse-grained version. FG-Trap saves around 6–23% of total energy, while FG-SyC saves even more, ranging from 11% to 28% for both

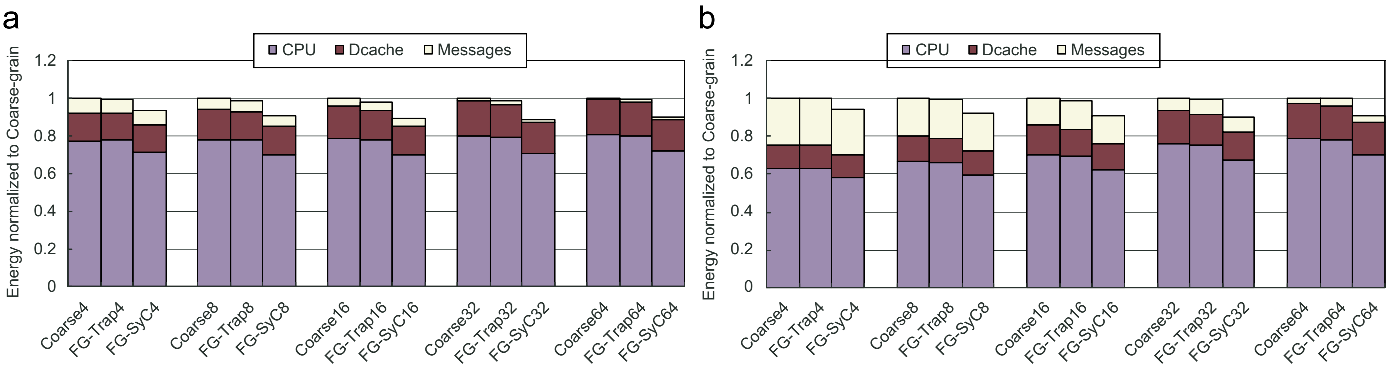

Fig. 13. Energy consumption for LU: (a) message cost as 64X of L1 cache energy cost; (b) message cost as 256X of L1 cache energy cost.

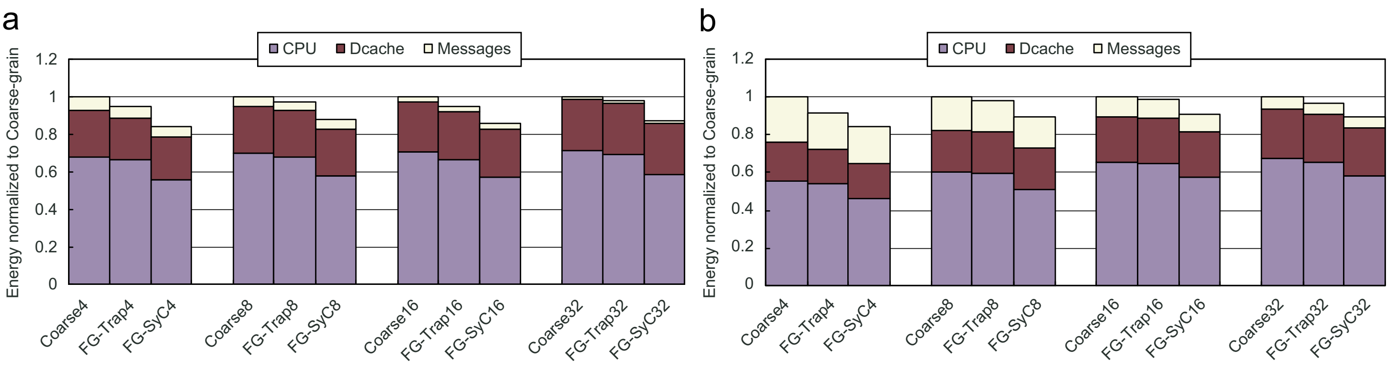

Fig. 14. Energy consumption for MST: (a) message cost as 64X of L1 cache energy cost; (b) message cost as 256X of L1 cache energy cost.

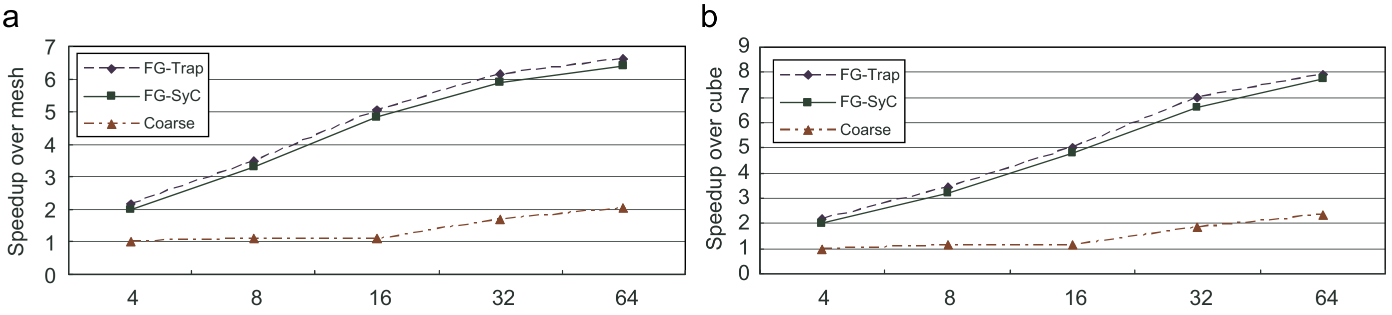

Fig. 15. Performance of MICCG3D: (a) on a mesh network; (b) on a hypercube.

64X and 256X cases. As discussed, the savings for FG-SyC come from elimination of busy waiting and the reduction in the number of messages compared to the FG-Trap.

The energy results for the application LU are shown in Fig. 13. The results show that FG-Trap does not improve energy compared to Coarse. However, FG-SyC saves 6–10% for both the 64X and 256X cases, compared to both FG-Trap and Coarse, due to the removal of busy waiting and reduction of messages. This is an example where FG-SyC reduces energy consumption significantly although it does not achieve much performance speedup over FG-Trap.

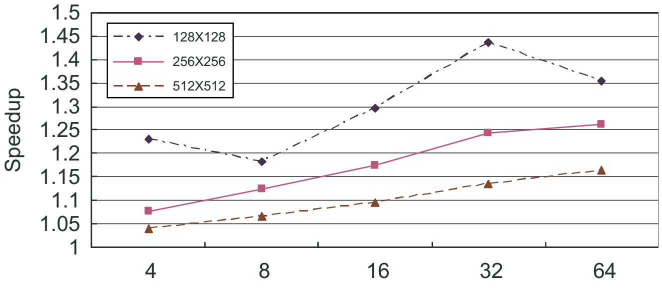

Fig. 16. Speedup of LU with different data sizes.

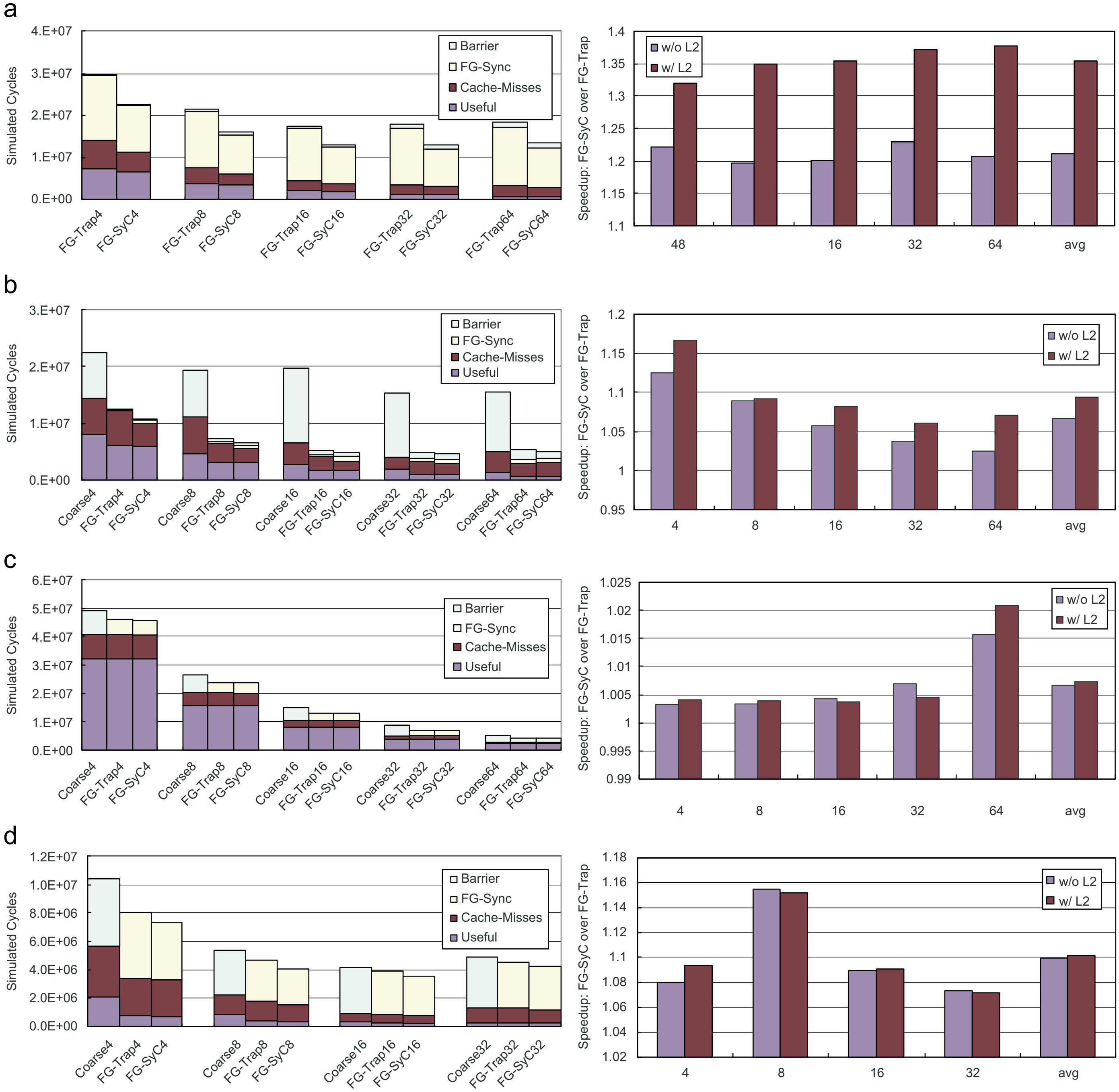

Fig. 17. Execution time of the applications with perfect L2 caches and speedup comparison with the case of no L2. (a) DNA chain, (b) MICCG3D, (c) LU, (d) MST.

Fig. 14 presents the energy comparison for MST. FG-Trap only shows slight improvement compared to Coarse (less than 5%), while FG-SyC achieves up to 15% overall energy savings due to the same reason as LU.

5.3. Sensitivity analysis

5.3.1. Interconnect network

First, let us look at how the network configuration affects the performance. Fig. 15 shows the performance speedup obtained for MICCG3D with different synchronization mechanisms and underlying interconnect configurations. Fig. 15(a) shows the speedup obtained on a bi-directional wrap-around mesh network, and Fig. 15(b) shows the speedup on a hypercube. We can see that the hypercube version is more scalable, resulting in a higher speedup, since it has less communication cost. However, we also find that the speedup of “FG-SyC” over “FG-Trap” are slightly less in a hypercube network, with an average of 5%. This is expected, since a hypercube network has lower message cost (that would dilute the improvement of FG-SyC due to the reduction of network messages).

5.3.2. Input data size

Fig. 16 shows the speedup (FG-SyC vs. Coarse) of LU on different data sizes. First, we can see that for all three data sizes, except 128 × 128, the speedup is increased as the number of processors grows. This is because the synchronization overhead becomes larger as the machine size increases and therefore FG-SyC shows more advantage. Across different data sizes on the same number of processors, the speedup becomes smaller as the data size increases. This is because for larger data sizes on a given configuration, the execution time is dominated by computation rather than synchronization and the benefits of synchronization optimizations are diluted.

5.3.3. L2 cache

Due to the complexity of the implementation, we did not implement a fully functional coherent L2 cache; instead in this section we try to predict how the introduction of an L2 cache per node might affect the above presented results for the FG-SyC approach.

To show the impact of L2 caches, we first show the results on SyC based on a perfect L2 cache with no cache misses; cache coherence cost is also ignored. The performance numbers, when there is an L2 cache of a fixed size, should fall between the numbers presented before with no L2 and the numbers presented here.

Fig. 17 shows the results with an 18-cycle L2 cache with 100% cache hit rate. In the figure, for each application, we present the breakdowns of the execution time as previously shown, and a comparison of the speedup achieved by FG-SyC over FG-Trap. We can see that the speedup is typically better for all four applications, compared to the numbers presented earlier without L2.

For example, for the DNA chain application, we achieve a speedup of 32–38%, with an average of 35%, compared to the average of 21% speedup achieved by FG-SyC with no L2. For MICCG3D, the average speedup of FG-SyC over FG-Trap is up from 6.7% to 9.4%. The speedup numbers are also slightly better for LU and MST on average, although the difference is less than 1% for these two benchmarks.

The reason for the difference is that, with L2 caches, the execution time spent on cache misses is significantly lower; however, a major part of performance improvement from FG-SyC comes from reducing the trap related time of FG-Trap, which basically stays the same regardless of L2 caches. Thus the speedup numbers improve because the total execution time is reduced with L2 caches. Although the actual numbers will depend on the real L2 design and implementation, as well as the characteristics of the applications, systems with L2 caches would have speedups better than the results shown without L2 and worse than with the perfect L2 caches shown in this section.

6. Conclusion

Fine-grained synchronization is a valuable mechanism for speeding up parallel execution by avoiding false data dependencies and unnecessary waiting. This paper has presented and evaluated Synchronization Coherence (SyC)—a novel approach, which integrates fine-grained synchronization with cache coherence. SyC is in fact a cache coherence mechanism for a full/empty-tagged shared memory that treats synchronization misses as cache misses and enables synchronized memory operations to be executed transparently without traps. We have presented architectural support and described the SyC protocol. We have shown that SyC requires minimal hardware extension to cache coherence, and it is transparent to processor nodes.

We have shown that SyC has significant advantage on both performance and energy consumption, compared with traditional and previous fine-grained synchronization mechanisms. To evaluate our approach, we have developed a complete simulation and compilation flow, and have conducted extensive simulation for a large spectrum of system configurations. For the applications studied, SyC improves overall performance by up to 23% over the previously published fine-grained synchronization approach. Systems that have L2 caches would benefit further: our experiments indicate a speedup potential of up to 38% for the applications studied. Our simulation results also show that SyC improves chip-wide energy consumption by up to 24%, by reducing busy waiting and network traffic.

Acknowledgment

This work was supported in part by NSF under Grant CCF-0105516, a research grant from the Swedish Foundation for International Cooperation in Research and Higher Education (STINT) (Vlassov), and NSF CCR-0311180 (Weiss). We also want to thank Zhenghua Qi for his involvement in the development of the simulator and all the anonymous reviewers for their useful comments and suggestions.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, F.T. Chong, K.L. Johnson, D. Kranz, J.D. Kubiatowicz, B.-H. Lim, K. Mackenzie, D. Yeung, The MIT Alewife machine, Proc. IEEE, Special Issue on Distributed Shared Memory 87 (3) (1999) 430–444.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, B. Smith, The tera computer system, in: ICS '90: Proceedings of the Fourth International Conference on Supercomputing, ACM Press, New York, NY, USA, 1990, pp. 1–6.

[3] Arvind, R.S. Nikhil, K.K. Pingali, I-structures: data structures for parallel computing, ACM Trans. Program. Lang. Syst. 11 (4) (1989) 598–632.

[4] P.S. Barth, R.S. Nikhil, Arvind, M-structures: Extending a parallel, non-strict, functional language with state, in: J. Hughes (Ed.), FPCA, Lecture Notes in Computer Science, vol. 523, Springer, Berlin, 1991, pp. 538–568.

[5] J.L. Bently, A parallel algorithm for constructing minimum spanning trees, J. Algorithm. 1 (1) (1980) 51–59.

[6] D. Brooks, V. Tiwari, M. Martonosi, Wattch: a framework for architectural-level power analysis and optimizations, in: ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture, ACM Press, New York, NY, USA, 2000, pp. 83–94.

[7] D. Burger, T.M. Austin, The SimpleScalar tool set, version 2.0, SIGARCH Comput. Archit. News 25 (3) (1997) 13–25.

[8] M.C. Carlisle, A. Rogers, Software caching and computation migration in Olden, in: PPOPP '95: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM Press, New York, NY, USA, 1995, pp. 29–38.

[9] M. Cintra, J.F. Martinez, J. Torrellas, Architectural support for scalable speculative parallelization in shared-memory multiprocessors, in: ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture, ACM Press, New York, NY, USA, 2000, pp. 13–24.

[10] M. Cintra, J. Torrellas, Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors, in: HPCA '02: Proceedings of the Eighth International Symposium on High-Performance Computer Architecture, IEEE Computer Society, Washington, DC, USA, 2002, p. 43.

[11] K.I. Farkas, N.P. Jouppi, Complexity/performance tradeoffs with non-blocking loads, in: ISCA '94: Proceedings of the 21st Annual International Symposium on Computer Architecture, IEEE Computer Society Press, Los Alamitos, CA, USA, 1994, pp. 211–222.

[12] L. Hammond, M. Willey, K. Olukotun, Data speculation support for a chip multiprocessor, in: ASPLOS-VIII: Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, ACM Press, New York, NY, USA, 1998, pp. 58–69.

[13] M. Horowitz, M. Martonosi, T.C. Mowry, M.D. Smith, Informing memory operations: memory performance feedback mechanisms and their applications, ACM Trans. Comput. Syst. 16 (2) (1998) 170–205.

[14] A. Kagi, D. Burger, J.R. Goodman, Efficient synchronization: let them eat QOLB, in: ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture, ACM Press, New York, NY, USA, 1997, pp. 170–180.

[15] S.W. Keckler, W.J. Dally, D. Maskit, N.P. Carter, A. Chang, W.S. Lee, Exploiting fine-grain thread level parallelism on the mit multi-alu processor, in: ISCA '98: Proceedings of the 25th Annual International Symposium on Computer Architecture, IEEE Computer Society, Washington, DC, USA, 1998, pp. 306–317.

[16] D. Kim, D. Yeung, Design and evaluation of compiler algorithms for pre-execution, in: ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM Press, New York, NY, USA, 2002, pp. 159–170.

[17] D. Kranz, B.-H. Lim, A. Agarwal, D. Yeung, Low-cost support for fine-grain synchronization in multiprocessors, in: R.A. Iannucci (Ed.), Multithreaded Computer Architecture: A Summary of the State of the Art, Kluwer Academic Publishers, Dordrecht, 1994, pp. 139–166, (Chapter 7).

[18] J. Laudon, D. Lenoski, The SGI Origin: a ccnuma highly scalable server, in: ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture, ACM Press, New York, NY, USA, 1997, pp. 241–251.

[19] B.-H. Lim, A. Agarwal, Reactive synchronization algorithms for multiprocessors, in: ASPLOS-VI: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, ACM Press, New York, NY, USA, 1994, pp. 25–35.

[20] N. Manjikian, Multiprocessor enhancements of the SimpleScalar tool set, SIGARCH Comput. Archit. News 29 (1) (2001) 8–15.

[21] J.F. Martinez, J. Torrellas, Speculative synchronization: applying thread-level speculation to explicitly parallel applications, in: ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM Press, New York, NY, USA, 2002, pp. 18–29.

[22] C.A. Moritz, M.I. Frank, LogPC: modeling network contention in message-passing programs, IEEE Trans. Parallel Distrib. Syst. 12 (4) (2001) 404–415.

[23] J.T. Oplinger, D.L. Heine, M.S. Lam, In search of speculative thread-level parallelism, in: PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques, IEEE Computer Society, Washington, DC, USA, 1999, p. 303.

[24] M.K. Prabhu, K. Olukotun, Using thread-level speculation to simplify manual parallelization, in: PPoPP '03: Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM Press, New York, NY, USA, 2003, pp. 1–12.

[25] R. Rajwar, J.R. Goodman, Speculative lock elision: enabling highly concurrent multithreaded execution, in: Proceedings of the 34th Annual International Symposium on Microarchitecture, IEEE Computer Society TC-MICRO and ACM SIGMICRO, Austin, Texas, 2001, pp. 294–305.

[26] R. Rajwar, J.R. Goodman, Transactional lock-free execution of lock-based programs, in: ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM Press, New York, NY, USA, 2002, pp. 5–17.

[27] L. Rauchwerger, D.A. Padua, The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization, IEEE Trans. Parallel Distrib. Syst. 10 (2) (1999) 160–180.

[28] M.C. Rinard, Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives, ACM Trans. Comput. Syst. 17 (4) (1999) 337–371.

[29] P. Shivakumar, N. P. Jouppi, CACTI 3.0: an integrated cache timing, power, and area model, Technical Report WRL-2001-2, Hewlett Packard Laboratories, December 28, 2001.

[30] B.J. Smith, Architecture and applications of the HEP multiprocessor computer, in: Real-Time Signal Processing IV, vol. 298, 1981, pp. 241–248.

[31] J.G. Steffan, C.B. Colohan, A. Zhai, T.C. Mowry, A scalable approach to thread-level speculation, in: Proceedings of the 27th Annual International Symposium on Computer Architecture, IEEE Computer Society and ACM SIGARCH, Vancouver, British Columbia, 2000, pp. 1–12.

[32] D.M. Tullsen, J.L. Lo, S.J. Eggers, H.M. Levy, Supporting fine-grained synchronization on a simultaneous multithreading processor, in: Proceedings of the Fifth International Symposium on High-Performance Computer Architecture, IEEE Computer Society TCCA, Orlando, FL, 1999, pp. 54–58.

[33] T.N. Vijaykumar, S. Gopal, J.E. Smith, G. Sohi, Speculative versioning cache, IEEE Trans. Parallel Distrib. Syst. 12 (12) (2001) 1305–1317.

[34] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, A. Agarwal, Baring it all to software: Raw machines, Computer 30 (9) (1997) 86–93.

[35] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, The SPLASH-2 programs: characterisation and methodological considerations, in: Proceedings of the 22nd Annual International Symposium on Computer Architecture, ACM Press, New York, 1995, pp. 24–37.

[36] S.C. Woo, J.P. Singh, J.L. Hennessy, The performance advantages of integrating block data transfer in cache-coherent multiprocessors, in: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society, San Jose, CA, 1994, pp. 219–229.

[37] J.-S. Yang, C.-T. King, Designing tree-based barrier synchronization on 2d mesh networks, IEEE Trans. Parallel Distrib. Syst. 9 (6) (1998) 526 –534.

[38] D. Yeung, A. Agarwal, Experience with fine-grain synchronization in MIMD machines for preconditioned conjugate gradient, in: PPoPP'93—Symposium on Principles and Practice of Parallel Programming, 1993, pp. 187–197.

[39] M. Zhang, K. Asanovic, Highly-associative caches for low-power processors, in: Kool Chips Workshop, 33rd International Symposium on Microarchitecture, 2000.

Yao Guo is an assistant professor in the Operating Systems Lab of the School of Electronics Engineering and Computer Science at Peking University, China. He received his PhD degree in computer engineering from University of Massachusetts at Amherst in 2007. His research interests include operating systems, low-power design, compilers, embedded systems and software engineering.

Vladimir Vlassov is an associate professor in computer systems in the School of Information and Communication Technology at the Royal Institute of Technology (KTH), Stockholm, Sweden. He received his PhD in computer engineering in 1984 from the Electrotechnical University (LETI) of St. Petersburg, Russia, where he was an assistant professor and an associate professor during 1985–1993. In 1993, he joined KTH, Stockholm, Sweden. In 1998, Dr. Vlassov worked as a visiting scientist in the Laboratory for Computer Science, Massachusetts Institute of Technology (MIT), Cambridge, USA. In 2004, he worked as a researcher at the Electrical and Computer Engineering Department at the University of Massachusetts (UMASS), Amherst, USA. His current research interests include parallel and distributed computing, peer-to-peer and grid computing, computer architecture.

Raksit Ashok has a bachelor's degree in electronics from the University of Swaziland and a masters degree in computer engineering from University of Massachusetts, Amherst. His primary research interests include computer architecture/microarchitecture, optimizing compilers, and distributed computing. He currently works at Google, and previously has worked in the semiconductor and high performance computing industries as a computer engineer.

Richard Weiss is currently a faculty member of The Evergreen State College in Olympia, WA. He has been a research associate professor at the University of Massachusetts, Amherst, where he was a member of the computer vision lab. He also worked for Digital Equipment Corp and Compaq Computer Corp, where he did research on computer architecture. He received an AB, in Mathematics from Brandeis University in 1969 and a PhD in Mathematics from Harvard University in 1976.

Csaba Andras Moritz obtained the PhD degree in computer systems from the Royal Institute of Technology, Stockholm, Sweden, in 1998. He is a tenured associate professor in the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst. From 1997 to 2000, he was a research scientist at MIT, Laboratory for Computer Science. He has consulted for several technology companies in Scandinavia and held industrial positions ranging from CEO, to CTO, and to founder. His most recent startup company, BlueRISC Inc, develops security microprocessors and hardware assisted security solutions. His research interests include computer architecture, compilers, low-power design, security, and nanoscale systems.