

# Fast Synchronization for Chip Multiprocessors

Jack Sampson\*  
CSE Dept  
UCSD

Rubén González†  
Dept. of Comp. Arch.  
UPC Barcelona

Jean-Francois Collard, Norman P. Jouppi, Mike Schlansker  
Hewlett-Packard Laboratories  
1501 Page Mill Road  
Palo Alto, California

## ABSTRACT

This paper presents a novel mechanism for barrier synchronization on chip multi-processors (CMPs). By forcing the invalidation of selected I-cache lines, this mechanism starves threads and thus forces their execution to stop. Threads are let free when all have entered the barrier.

We evaluated this mechanism using SMTSim and report much better (and most importantly, more flat) performance than lock-based barriers supported by existing microprocessors.

## 1. INTRODUCTION

Chip multiprocessors may radically change the landscape of parallel processing by the fact they'll soon be ubiquitous, giving ISVs an incentive to multithread their applications. As some researchers have argued, they may also be simpler to program than classic multiprocessors because communications and synchronizations take place on die [2].

Still, some scientific applications require frequent synchronizations between relatively small amounts of computations, and are best expressed using a SIMD or vector style of programming. Indeed, there recently has been regained interest in vector accelerators, whether as co-processors or as PCI-Express attached acceleration cards.

The goal of this research is to study how well off-the-shelf CMPs are suited to fine-grain parallel processing, or if they could be a good fit after minimal modifications. (Major redesign is not an appealing option unless you can justify a large performance benefit or a huge market, or, probably, both.) As a consequence, a design constraint for us is to *not* modify the cores and, in particular, neither their pipelines nor their register files. Another constraint is to require no new instruction other than those provided by existing ISAs.

Under these constraints, and since current CMPs were designed with coarse grained parallelism in mind where threads seldom synchronize among each other, this paper tackles this specific question: Can we tweak a general-purpose CMP to provide extremely fast barrier synchronization among multiple threads of a single application?

---

\*While at HP Labs. Also funded by NSF grant No. CNS-0509546.

†While at HP Labs.

## 2. A NEW SYNCHRONIZATION METHOD

### 2.1 Overview

This paper leverages one key property of all existing, unmodified cores: when the next instruction to be executed is fetched and the fetch misses in the I-cache, the thread will stall indefinitely when it is this instruction's turn to execute. The thread will resume execution when the cache line comes back and the instruction can be read. Resuming execution is extremely fast – in fact, reading an instruction from the first-level I-cache is one of the fastest thing a core can do, often in one cycle.

We achieve global (barrier) synchronization within a CMP by invalidating distinguished I-cache lines that contain the execution point at which the threads should synchronize. Some additional logic, typically placed in the L2 cache control, filters fill requests for those cache lines and refuses to serve them until a specific condition is met. While this condition is not met, threads waiting for the I-cache lines stall on their current PC. The filter logic freezes fill requests for that address until all threads are stalling on their distinguished I-cache line; when this condition is met, the filter knows all threads have entered the barrier and that all threads can be freed; to do so, the filter completes the fill requests.

More precisely, our barrier method works as follows: the application executed by the threads contains a call to `barrier()`. By construction, a portion of the `barrier()` text is aligned on an I-cache line boundary; the address of that line is entered in an entry in the filter. When a thread starts executing the code of `barrier()`, it will eventually attempt to fetch this line; the fetch will miss, and the cache line will be blocked by the filter until all other threads have reached that point.

Once a thread's fill request is serviced, execution resumes normally and instructions in the provided I-cache line are fetched by the pipeline. The threads have already synchronized at this point, so the threads then just return from the call to `barrier()`.

### 2.2 Detailed Implementation

In this section, we detail a more realistic implementation that still makes several assumptions: first, the filter is under OS control and, in particular, saved and restored when an application is de-scheduled or the page holding the program text of the barrier is swapped out. Second, we assume there

are as many threads as cores. Third, we assume fill requests to filtered addresses are not coalesced by hardware before reaching a filter. Fourth, we assume that any trace cache line that contains micro-ops from an invalidated cache line is flushed or invalidated. We come back to some of these assumptions in Section 3.

The code of `barrier()` consists of two parts, head and tail. The goal of head is to invalidate the cache line (one line is enough) that contains the tail; the goal of the tail is to implement the barrier — that is, the tail is contained in the distinguished I-cache line. The program text of the barrier is assumed to be aligned to cache lines of the first level I-cache. The size,  $L$ , of these lines is typically smaller than that of outer cache levels and line inclusion is preserved with respect to outer cache levels. The actual code for our method, as used in our simulations, is shown in Appendix B. (The code for a standard barrier implementation is provided in Appendix A.)

Let  $A$  be the address at which the program text of `barrier()` begins, that is, the head’s starting address. The *second* cache line of program text contains the tail and its address is  $A+L$ . This line is the distinguished line on which threads will synchronize. This line is initially invalidated by the filter. The filter is initially in the state where it blocks all fill requests for address  $A+L$ . All observed but blocked fill requests are stored and serviced later.

The first cache line of program text in the procedure (which therefore is at address  $A$ ) contains two instructions: one that invalidates the next line at address  $A+L$ , and one that discards prefetched instructions. Explicit invalidations by software can be done using the `fc` instruction on Itanium or the `ICBI` instruction on the PowerPC architecture. These invalidations are propagated throughout the cache hierarchy, and we assume they are passed to the filter by the innermost cache. The filter does not propagate the invalidations. These invalidations purge copies of the distinguished cache line from cache levels between the core and the filter, making sure the thread will stall on fetching line  $A+L$ . The second instruction makes sure no prefetched copy of the instruction is kept internally by the processor. Discarding prefetched instructions is provided, for example, by the PowerPC `ISYNC` instruction.

The filter counts the invalidates it receives for the distinguished address, and when all threads have invalidated the line at address  $A+L$  (i.e., the barrier tail), the filter enters the state where it starts servicing fill requests (both the requests that are coming up and those that were blocked and are pending). (Note that we assume instructions won’t be invalidated explicitly except by our barrier mechanism; the line may be evicted from the cache, but silently.) Until then, fill requests for address  $A+L$  were blocked; if that cache line had been prefetched by hardware, the prefetch could not trigger an early opening of the barrier: the barrier only opens when all threads have explicitly said they entered the barrier using the invalidate instruction.

Coming back to threads, note that the invalidate instructions are state-changing and therefore will not be speculatively committed by speculative execution hardware. After

these instructions are done, the thread’s PC moves on to address  $A+L$  and experiences a cache miss. The thread’s pipeline stalls until the miss is serviced.

When all threads have reached this point, the missing line at address  $A+L$  is provided by the filter. This line is the barrier tail; it contains a command to tell the filter that the current thread is exiting the barrier; from then on, fill requests for address  $A+L$  *coming from this thread* won’t be serviced by the filter anymore, until the filter reaches the “service” state again. This takes care of the case where one thread runs far ahead of the others; for example, one thread may reach another barrier (or another instance of the same static call to `barrier()`) before other threads have even exited the current barrier. Again, the first thing that threads do on barrier exit is to send a command to the filter to indicate they are exiting, so a thread that runs ahead will be starved when its PC reaches  $A+L$  again.

This command to the filter can be implemented in various ways; the one we have been evaluating is symmetric to barrier entry: each thread invalidates another, agreed-upon I-cache line that contains dead code. The address of that line is denoted by  $E$ . The identity of the core making the invalidation is typically carried with the request, allowing the filter to know which core it should stop servicing requests from.

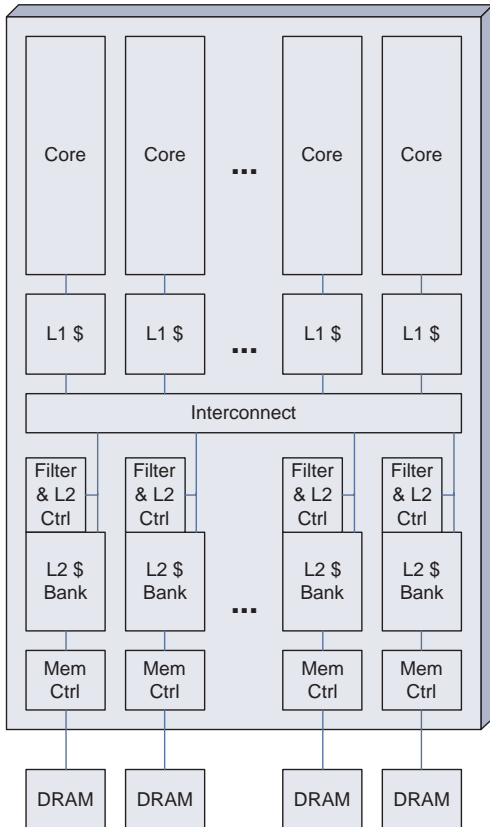
Finally, the text for the `barrier()` procedure ends with a procedure return instruction, possibly followed by nops to pad the tail up to the next first-level cache line boundary. The complete code for this implementation of `barrier()` is provided in Appendix B.

When the filter sees that all threads have exited the barrier, it goes back to the initial state where it services no fill request at address  $A+L$  and expects explicit invalidates at that address. Note that, because one thread may run ahead, the filter may observe invalidates of  $A+L$  for the next barrier before all exit commands for the current barrier have been received.

## 2.3 Filter Microarchitecture & Operation

Figure 1 shows a representative CMP organization, where each core has a private L1 cache and accesses a shared L2 spread over multiple banks. This abstract organization was also selected in [4]. The number of banks does not necessarily equal the number of cores: The Power5 processor sports two cores, each providing support for two threads, and its L2 consists of 3 banks [7]; the Niagara processor offers 8 cores supporting 4 threads each, and features a 4-banked L2 cache [8]. In Niagara, the interconnect linking cores to L2 banks is a crossbar.

The novel aspect in Figure 1 lies in the replicated filter incorporated into the L2 cache controllers. The filter replicas are in fact identical, but a single copy handles all the traffic related to a given barrier. The filter is tightly integrated with the L2 controller: on a fill request, the controller can quickly tell from the target address if the request is a hit. If it is, and the address matches a filter entry, then the line is marked as blocked. Likewise, if the request is a miss, its status is marked as blocked so that it isn’t provided to the



**Figure 1: Organization of a standard multicore augmented with a replicated filter integrated with the L2 cache controller.**

core when the data returns from outer levels of the hierarchy. Observe that this organization puts the filter out of the critical path for incoming requests and outbound responses; the controller and the filter can check the status of a line based on its address faster than the data can be read, so our filtering mechanism does not increase the L2 latency.

Since memory requests are directed to memory channels depending on their physical target addresses, the barrier library must make sure that addresses A and E for a given barrier map to the same filter. The filter contains a table with a fixed number of entries, and a set of finite automata – one per entry. There is one table entry and one automaton per barrier supported by our mechanism at any given time. The OS is responsible for saving and restoring the content of an entry if it decides to use it for another application. Each application only needs a single code for `barrier()` and therefore a single filter entry.

Each table entry contains the address A of the barrier program text, the address E of the I-cache line used by threads to indicate they are exiting the barrier, a counter C, bit vectors PENDING and EXITED of size N (where N is the number of threads), storage for the content of the cache line, and a bit to indicate if the content of the storage is valid. A filter is initialized at the beginning of the application; OS support is needed to provide the physical address of the

head and tail parts of the program text of the barrier, i.e., physical addresses A and E. Once warmed up, an entry's storage will keep the content of the distinguished cache line and therefore acts as a barrier cache. The cache line storage may either be in the associated L2 bank or in the filter control.

On an invalidate, the filter checks whether the invalidate's target address equals the A or E field in one of its table's entries. On a fill request, the filter checks whether the target address equals the A field of one of its entries.

As illustrated in Figure 2, the finite automaton for a given barrier has two main states: the Blocking and Service states. When an entry is inserted in the table, the corresponding automaton starts in the Blocking state, its counter C and bit vectors PENDING and EXITED are set to zero, and the cache line storage is invalidated.

In the Blocking state, the filter processes an incoming fill request as follows: it checks if the content of the entry's storage is valid; if it isn't, it sends the request to memory and puts the returned cache line in the storage; in either case, it does not service the request – that is, the cache line is not passed to the requesting core. The request is marked as pending by setting the bit in PENDING that corresponds to the requester.

In this state, the filter also counts I-cache invalidates targeting address A using the C field of the corresponding table entry. When this counter reaches N, the filter resets C to zero and goes to an intermediate state where pending fill requests are serviced. Fill requests that arrive while the filter is in this state are serviced as well. Servicing a request consists of checking whether the content of the storage is valid; if it isn't, the cache line is read from memory and copied into storage; if it is, the line is provided to the requester. The filter then proceeds to the Service state.

In the Service state, the filter services incoming fill requests for address A. It also monitors invalidates targeting address E. All transactions going through the memory hierarchy normally carry the ID of the originating core; the filter uses that ID to identify which cores exited the barrier, and sets the corresponding EXITED bit. Note that each new request is serviced in the Service state if the sending core hasn't exited the barrier, that is, until its EXITED bit is set.

If setting a EXITED bit makes all EXITED bits equal to 1 (for the current entry), the filter goes back to the Blocking state and clears all EXITED bits. The content of the line storage doesn't need to be invalidated since it will be reused.

### 3. EXTENSIONS

As detailed earlier, our mechanism makes a number of assumptions. In particular, we assumed that exactly one thread participating in a barrier executes per core. One reason was to simplify thread counting by the filter: with this assumption, the filter is designed to wait for as many threads as there are cores. The second reason is related to identifying threads that enter and exit a barrier: with exactly one thread per core, we can use the originating core's ID, which is piggybacked on each memory request, to identify threads.

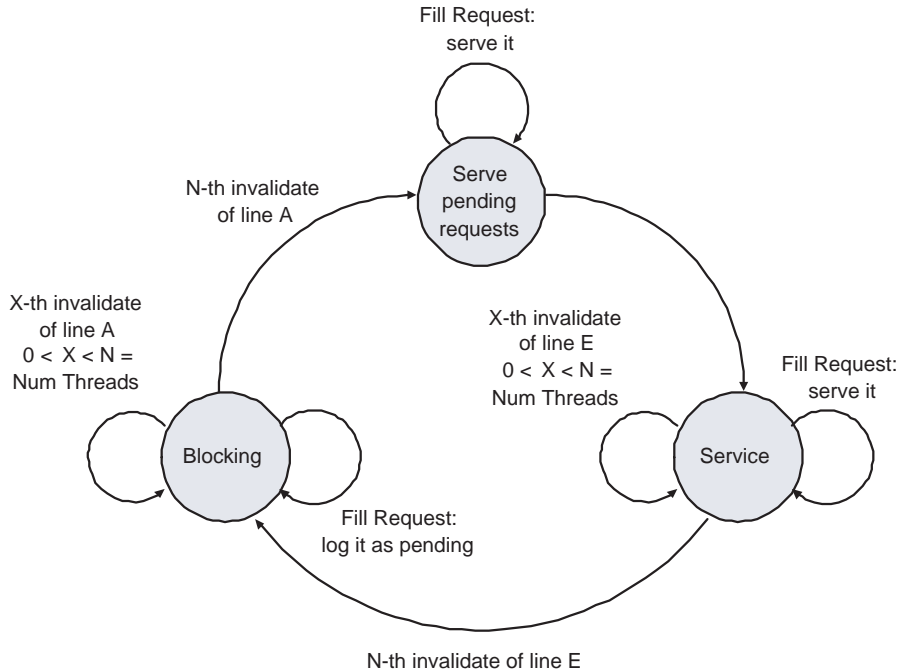


Figure 2: Finite State Automaton that implements the filter for a given entry and barrier.

The third reason relates to making sure that the “message” sent by a thread to the filter actually reaches the filter and is not merged with other messages. For example, processors usually coalesce reads, writes or invalidates that target the same address. The pitfall is that, if two threads run on the same core, and if both invalidate the same cache line at about the same time, the logic passing these requests up the cache hierarchy perceives one invalidate as superfluous and decides to either drop one or merge the two. This would prevent the filter from being alerted that both threads have entered (or exited) the barrier. (Note that a similar pitfall would appear if threads were communicating with the filter through stores to memory locations mapped to the filter’s control registers.)

Extending our mechanism to allow for *fewer* threads than there are cores is easy: filter entries are simply augmented with a register storing how many threads participate in the barrier, and the content of this register is copied into counter C when the last thread exits the barrier. Likewise, the value of that register controls how many bits in EXITED are to be used.

Extending our mechanism to accommodate *more* threads than there are cores is more complicated, however: On the software side, a distinguished I-cache line is assigned to exactly one thread; i.e., each thread stalls on its own distinguished I-cache line when entering the barrier, and it invalidates another reserved line to indicate it is exiting the barrier. Using distinct cache lines defeats request coalescing, and also serves to identify threads. These I-cache lines are unrelated, except that their addresses are optionally restricted to make their encoding in the filter easy — see below. The code shown in Appendix B thus has as many copies

as there are threads, and on a call to `barrier()`, a thread jumps to its private copy of Appendix B through some trampoline code (containing in essence a computed goto).

On the hardware side, the filter is modified as follows. Field A becomes an address pattern as opposed to a fixed address; given some implicit mask M decided at design-time, an invalidate operation targeting address X is considered as a message from the barrier library to the filter if  $X \text{ AND } M$  equals A. Typically, A specifies the highest-order bits so as to distinguish the text of the barrier library from the rest of the code, and a few low-order bits to facilitate the alignment of the barrier’s text. Likewise, field E becomes an address pattern used for invalidates on exit.

#### 4. RELATED WORK

Hardware implementations of barriers have been around for a long time; most conceptually rely on a wired-AND line connecting cores or processors [3, 6]. The most recent incarnation of this idea may be the global interrupt bus in Blue Gene/L [1, 5]. Other relevant work includes the CM-5’s control network [9] and the lock box of [12].

Dedicated interconnection networks may also have special-purpose cache hardware to maintain a queue of processors waiting for the same lock (see [10] and references within). The principal purpose of these hardware primitives is to reduce the impact of busy waiting. In contrast, our mechanism does not perform any busy waiting, nor does it rely on locks. In fact, it doesn’t perform any remote reference at all.

Hardware barriers on SoCs are addressed in [11]. That work offers fast atomic access to lock variables via a dedicated hardware unit. When a core fails to acquire a lock, its

Fetch width	4
Issue / Decode / Commit width	3 / 4 / 4
RUU size (Inst. window- ROB)	64
L1 DCache (one per core)	64kB, 2 ways, 1 cycle
L1 ICache (one per core)	64kB, 2 ways, 1 cycle
L2 Unified Cache (shared)	512 kB, 2 ways, 14 cycles
L3 Unified Cache (shared)	4096 kB, 2 ways, 38 cycles
Memory Latency	138 cycles
Filter (new design only)	Accepts 1 request per cycle

Table 1: Baseline configuration of the multicore.

request is logged in the hardware unit. When the lock is released, an interrupt will be generated to notify the core, which makes us believe that barriers using our scheme will be faster. ([11] does not report barrier timings.)

## 5. EXPERIMENTS

We have been using an unofficial version of SMTSim kindly provided by Jeff Brown and Dean Tullsen at UCSD. SMT-Sim does not support multiple threads spawned by the same application, i.e., it does not support threads sharing an address space. We therefore had to add that support to SMT-Sim.

SMTSim simulates multicores that obey the Alpha architecture, and we used that instruction set with the addition of the PowerPC ICBI and ISYNC instructions. We simulated CMPs with 2, 4, and 8 cores and thus as many threads. We focus our examination only on synchronizations occurring among threads executing on a single CMP; barriers involving multiple CMPs could be constructed in hierarchical fashion with additional software or hardware support. Note also that the filter doesn't have to be located in front of L2. Placing it further from the cores may cause barriers to take longer, but may on the other hand simplify chip design. We assumed a symmetric CMP, i.e., all cores are identical. Other simulation parameters are listed in Table 1.

Our simulations follow the methodology described in [6]: performance is measured as average time per barrier over a loop of four consecutive barriers with no work or delays between them, with the loop being executed 100,000 times.

Our results are shown in Figure 3. We compared our method with a standard centralized sense-reversal barrier based on locks, and with a binary combining-tree of such barriers [6]. Clearly, our method performs best by far, and, most importantly, scales extremely well. The code for the standard barrier is provided in Appendix A. The lock is implemented using the load locked (`ldq_l`) and store conditional (`stq_c`) instructions. Note that this simple method has been reported to be faster than or as fast as ticket and array-based locks [6]. Care was taken to place shared variables (such as the counter and the flag) in separate cache lines to avoid generating useless coherence traffic.

## 6. CONCLUSION

We presented a mechanism for barrier synchronization that does not rely on busy waiting on a shared variable, locks, or for that matter any coherence traffic. Instead, it relies on additional logic that blocks specific I-cache fills and thus allows us to starve threads at synchronization points. Because no coherence traffic is required, the performance obtained by our method is essentially flat with respect to the number of cores, which could pave the way for efficient fine-grain parallelism on minimally modified multicores.

## Appendix A: Code for the baseline barrier implementation

```
# Reg $16 = #of threads
# Reg $17 = &sense_var (thread-local)
# Reg $18 = &counter
# Reg $19 = &flag

Barrier:
    mb
    ldq $8,0($17)    # load local_sense
    not $8,$8        # toggle local_sense
    stq $8,0($17)    # set local_sense
    addq $31,1,$6    # set $6 to 1
Counter_decrement:
    ldq_l $0,0($18)  # load counter
    subq $0,$6,$0    # decrement counter
    mov $0,$1        # copy counter
    stq_c $1,0($18)  # try to store counter
    beq $1, Counter_decrement # retry if fail
    beq $0, Last_through
NotLast:
    ldq $0, 0($19)
    subq $0,$8,$0
    bne $0, NotLast
    br Exit_barrier
Last_through:
    stq $16,0($18)   # reset counter to num threads
    stq $8,0($19)    # Flag = local_sense
Exit_barrier:
    ret
```

## Appendix B: Code for our new barrier implementation

```
# Reg $17 = &InvalidateMe (A+L in the text)

# The head begins here
Barrier:                                # At address A
    ICBI $17
    ISYNC $17
    # Possibly nops to pad up to line boundary
# ---- L1 I-cache line boundary
# The tail begins here
InvalidateMe:                            # At address A+L
    ICBI $18
    ret                                    # return from barrier
    # Possibly nops to pad up to line boundary
```

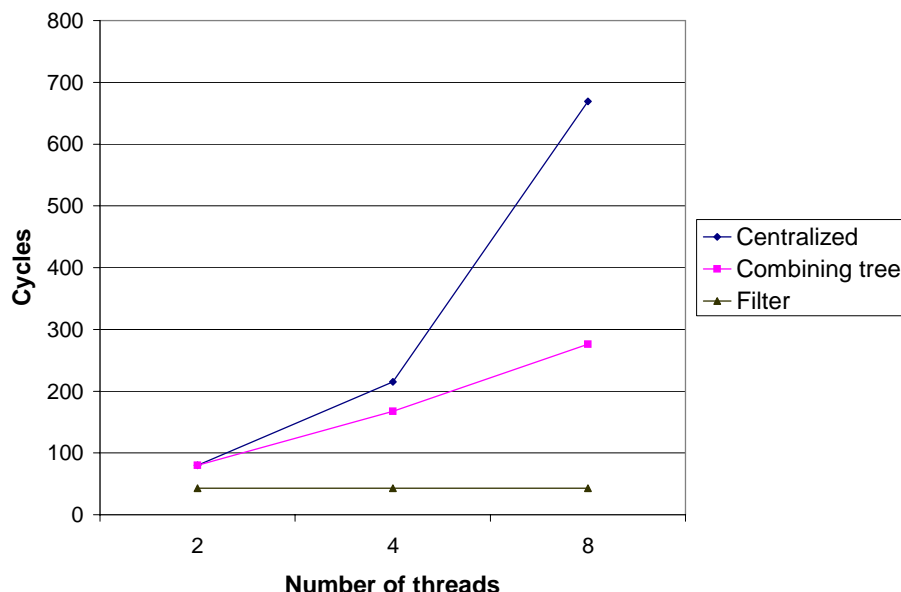


Figure 3: Average execution time of three different barrier mechanisms.

## 7. REFERENCES

- [1] G. Almasi et al. Design and implementation of message-passing services for the Blue Gene/L supercomputer. *IBM Journal of Research and Development*, 49(2/3):393–406, Mar. 2005.
- [2] S. Amarasinghe. Multicores from the compiler’s perspective: A blessing or a curse? Keynote at CGO’05, San Jose, CA. March 05.
- [3] C. J. Beckman and C. D. Polychronopoulos. Fast barrier synchronization hardware. In *Proc. Conf. on Supercomputing*, pages 180–189, 1990.
- [4] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-performance throughput computing. *IEEE Micro*, 25(3):32–45, May 2005.
- [5] P. Coteus et al. Packaging the Blue Gene/L supercomputer. *IBM Journal of Research and Development*, 49(2/3):213–248, Mar. 2005.
- [6] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture*. Morgan Kaufmann.
- [7] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 chip: a dual-core multithreaded processor. *IEEE Micro*, pages 40–47, March–April 2004.
- [8] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, Mar. 2005.
- [9] C. E. Leiserson et al. The network architecture of the Connection Machine CM-5. In *Proc. of SPAA*, pages 272–285, June 1992.
- [10] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Comp. Sys.*, 9(1):21–65, Feb. 1991.
- [11] B. E. Saglam and V. J. Mooney. System-on-a-chip processor synchronization support in hardware. In *Proc. of Conf. on Design, automation and test in Europe*, pages 633–641, Munich, Germany, 2001.
- [12] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proc. Int’l Symp on High-Performance Architecture (HPCA)*, Jan. 1999.