Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers

Jack Sampson*
CSE Dept. UC San Diego

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

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

Brad Calder
CSE Dept. UC San Diego
and Microsoft

Abstract

We examine the ability of CMPs, due to their lower on-chip communication latencies, to exploit data parallelism at inner-loop granularities similar to that commonly targeted by vector machines. Parallelizing code in this manner leads to a high frequency of barriers, and we explore the impact of different barrier mechanisms upon the efficiency of this approach.

To further exploit the potential of CMPs for fine-grained data parallel tasks, we present barrier filters, a mechanism for fast barrier synchronization on chip multi-processors to enable vector computations to be efficiently distributed across the cores of a CMP. We ensure that all threads arriving at a barrier require an unavailable cache line to proceed, and, by placing additional hardware in the shared portions of the memory subsytem, we starve their requests until they all have arrived. Specifically, our approach uses invalidation requests to both make cache lines unavailable and identify when a thread has reached the barrier. We examine two types of barrier filters, one synchronizing through instruction cache lines, and the other through data cache lines.

1. Introduction

Large-scale chip multiprocessors will radically change the landscape of parallel processing by soon being ubiquitous, giving ISVs an incentive to exploit parallelism in their applications through multi-threading.

Historically, most multi-threaded code that exploits parallelism for single-application speedup has targeted multi-chip processors, where significant inter-chip communication latency has to be overcome, yielding a coarse-grained parallelization. Finer-grained data parallelization, such as over inner-loops, has more traditionally been exploited by vector machines. Some scientific applications have dependencies between relatively small amounts of computations, and are best expressed using a SIMD or vector style of programming [26]. These dependencies may be implicit stalls within a processing element on a vector machine. In partitioning these calculations across multiple cores these dependencies require explicit synchronization, which may take the form of barriers. With chip multi-processors, the significantly faster communication possible on a single chip makes possible the implementation of very fast barrier synchronization. This will open the door to a new level of parallelism for multi-threaded code, exploiting smaller code regions on many-core CMPs. The goal of our research is to explore how to use many-core CMPs to exploit fine-grained data parallelism, and we focus primarily on vector calculations. However, our approach is more general, since it can work on some codes that may not be easily vectorizable.

To show the importance of fast barrier synchronization, consider Table 1, which shows the best speedups achieved by software-only barriers when distributing kernels across a 16-core CMP. For the Livermore loops, performance numbers shown are for vector lengths of 256; when distributed across 16 cores (in 8-element chunks for good performance with 64B cache lines), at a vector length of 256, all cores are kept busy, with each CMP core working on one (Livermore loop 2) or more chunks per iteration. As can be seen from the table, speedups using software-only approaches are not always present. This lack of speedup is primarily due to the relatively high latency of the software barriers in comparison to the fine grain parallelism of the distributed computation. In contrast, the approach we will describe always provides a speedup for the parallelized code for all of the benchmarks.

In this paper, we study the role of fast barriers in enabling the distribution and synchronization of fine-grained data parallelism on CMPs. We examine the impact of different barrier implementations on this manner of parallelization, and introduce *barrier filters*, a new barrier mechanism.

Our barrier filter design focuses on providing a solution that does *not* require core modification. This includes not modifying the pipeline, register file, nor the L1 cache, which is tightly integrated with the pipeline. Our design also does

*Started while at HP Labs. Also funded by NSF grant No. CNS-0509546.
†While at HP Labs.

Kernel	Best Software Barrier
Livermore loop 2	0.42
Livermore loop 3	1.52
Livermore loop 6	2.08
EEMBC Autocorrelation	3.86
EEMBC Viterbi	0.76

Table 1. Speedups and slowdowns achieved on kernels distributed across a 16 core CMP when using the best software barriers, in comparison to sequential versions of the kernels executing on a single core. Numbers less than 1 are slowdowns, and point to the sequential version of the code as being a better alternative to parallelism when using software barriers.

not require new instructions, as there are existing ISAs (e.g., PowerPC, IA-64) that already provide the necessary functionality. Instead, we leverage the shared nature of CMP resources, and modify the shared portions of the memory subsystem. The barrier filter relies on one intuitive idea: we make sure threads at a barrier require an unavailable cache line to proceed, and we starve their requests until they all have arrived.

In this paper we make the following contributions:

- We show the potential of CMPs as a platform well-suited to exploiting fine-grained data parallelism for vector computations.
- We introduce a new method for barrier synchronization, which will allow parallelism at finer granularities than traditionally associated with software barriers.
- The hardware support required by the method is less intrusive than other hardware schemes: no dedicated networks are needed, and changes are limited to portions of the memory subsystem shared among cores, making the method well suited to the way CMPs are designed today.

2. 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 [8]. Other combining networks for synchronization include multi-stage and single-stage shuffle-exchange networks [12] and wired-NOR circuits [22, 3]. Beckmann and Polychronopolous [3] assume a dedicated interconnect network for transmission of messages consisting of (Barrier ID, Barrier enable) and (Barrier ID, Set Processor Status) from each processor to a globally visible array of bit-vectors, each bit-vector corresponding to a barrier. For each barrier, zero-detect logic (wired-NOR) associated with each bit-vector then generates a signal upon the barrier condition being satisfied, passes the signal to a switch-box programmed by the last Barrier ID sent from each processor, and resets the global bit-vector associated with the satisfied barrier. The switch-box then distributes the signal, via an additional dedicated interconnect network, to the participating subset of processors.

Dedicated interconnect networks for barrier synchronizations are not uncommon among massively parallel computers, appearing in the Ultracomputer's fetch-and-add-combining switches [11], Sequent's synchronization's bus [2], the CM-5's control network [16], the Cray T3D [21], and the global interrupt bus in Blue Gene/L[1, 7]. Our work is primarily focused on much smaller scale systems, namely CMPs, with a more commodity nature. On a smaller scale system, although not commodity in nature, Keckler, et. al. [14] discuss a barrier instruction on the MIT Multi-ALU Processor that utilizes six global wires per thread to perform rapid barrier synchronizations between multiple execution clusters on the same chip. However, in contrast to all the above approaches, we do not require additional interconnect networks for our new barrier mechanism, instead we focus on using the existing memory interconnect network. We require neither additional processor state registers, nor that our core pipeline be subject to control from additional interrupt lines.

The Cray T3E [21], another massively parallel computer, abandoned the dedicated physical barrier network of its predecessor, the T3D, and instead allowed for the barrier/eureka synchronization units(BSUs), to be connected via a virtual network over the interconnect already used for communication between the processing nodes. The position of a particular BSU within a barrier tree was configurable via registers in the network router associated with a given node's BSUs. Processing nodes would communicate with a local BSU, which would send a barrier packet to their parent in the barrier tree when all of the BSU's children had signaled arrival, and forward release notifications from the BSU's parent to the children. Barrier packets were then given preferential routing priority over other network traffic, so as to mimic having their own network. Blocking and restarting of threads using the T3E barrier hardware requires either polling the status of the BSU or arranging to receive an interrupt when BSU status changes. Blocking and restarting of threads using our barrier filter mechanism is more lightweight, requiring neither polling of external registers nor the execution of an interrupt handler, instead directly stalling a processor through data starvation, and restarting via cache-fill.

Tullsen, et al. [24] examine hardware support for locks, but do not examine barriers, at both SMT (integrated with the load-store unit), and CMP (integrated with the shared L2 cache) levels. In contrast to this work, we do not introduce new synchronization primitives to access our hardware structures and alter pipeline flow, but instead make use of properties from existing instructions.

Dedicated interconnection networks may also have special-purpose cache hardware to maintain a queue of processors waiting for the same lock [10]. 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. By eliminating busy waiting, we can also reduce core power dissipation.

Saglam and Mooney [20] addressed hardware synchronization support on SoCs. That work offers fast atomic access to lock variables via a dedicated hardware unit. When a core fails to acquire a lock, its request is logged in the hardware unit. When the lock is released, an interrupt will be generated to notify the core. ([20] does not report barrier timings.)

There is continued interest in improving the performance of software based barriers, especially on large multiprocessor and multicomputer systems, where traditional hardware based solutions may encounter implementation cost, scalability, or flexibility issues [18]. Nikolopolous et al. [19] improve the performance of existing algorithms by using hardware primitives for uncacheable remote read-modify-write for the high-contention acquire phases of synchronization operations, obviating coherence traffic, while using cache-coherent operations for polling of the release condition. Cheng and Carter [5] exploit the increasing difference between local and remote references in large shared memory systems with a barrier algorithm tuned to minimize round-trip message latencies, and demonstrate the performance improvements possible through utilizing a coherence protocol that supports hardware write-update primitives. Our focus, unlike the above and most prior work concerning barriers, is on CMPs, rather than large collections of processors. We therefore have rather different scalability concerns, but the low latency of communication on CMPs allows us to explore finer grained parallelism than is traditionally exploited via thread level parallelism.

3. Barrier Filter

3.1. Barrier Filter Overview

The goal of our approach is to find a good performance vs. design cost trade-off for fast global barrier synchronization. Our design is based on the observation that all memory access requests that go through a cache stall until the cache fill occurs. We can extend this mechanism to stall later memory requests and thus perform synchronization. We can provide barrier synchronization for a set of threads by making those threads access specific cache lines, which are not filled until the criteria for the barrier occurs. A thread using this barrier mechanism proceeds through the following three steps: (1) a signaling step, to denote the thread's arrival at the barrier, (2) a synchronizing step, to make sure the thread does not continue to execute instructions after the arrival at the barrier, until the barrier access completes, and (3) upon resuming execution, a second signaling step, to denote that the thread has proceeded past the barrier.

We achieve global barrier synchronization by having each thread access a specific address, called an *arrival address*. Each thread is assigned a distinct arrival address by the operating system, which maps to a different cache line. A *barrier filter*, a hardware structure consisting of a state table and associated state machines, is placed in the controller for some shared level of memory, potentially even main memory. As increased distance from the core implies increased communication latency, we envision the most likely placement of the barrier filter to be in the controller for the first shared level of memory. The barrier filter keeps track of the arrival address assigned to each thread. The filter listens for invalidation requests for these arrival addresses, which signal that a thread has arrived at a barrier. After a thread has arrived at a barrier, it is blocked until all of the threads have arrived at the barrier.

To perform the barrier, each thread executes an instruction that invalidates the arrival address, and then attempts to access (load) the arrival address. The invalidate instruction removes any cache block containing this address from the cache hierarchy above the barrier filter, and indicates to the barrier filter that the thread has reached the barrier. The thread then does a fill request, which the barrier filter will filter and not service, as long as that thread is blocked at the barrier. This prevents the thread from making forward progress until all of the threads have arrived at the barrier. Once all of the threads have arrived, the barrier filter will allow the fill requests to be serviced.

After the fill request to the arrival address is resolved, a thread needs to inform the filter that it has proceeded past the barrier and is now eligible to enter a barrier again. One way to do so is to have the thread access a second address called an *exit address*.Thus, along with the arrival address, each thread is also assigned an exit address, which represents a distinct cache-line and is also kept track of in the barrier filter.

3.2. Barrier Filter Architecture

We now describe the barrier filter architecture and the baseline multi-core architecture we assume. 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 contains two cores, each providing support for two threads, and its L2 consists of 3 banks [13]; the Niagara processor has 8 cores, supporting 4 threads each, and a 4-way banked L2 cache [15]. In Niagara, the interconnect linking cores to L2 banks is a crossbar.

The barrier filter architecture is shown in Figure 1 to be incorporated into the L2 cache controllers at each bank. As memory requests are directed to memory channels depending on their physical target addresses, the operating system must make sure that all arrival and exit addresses it provides for a given barrier map to the same filter. For our design, the hardware can hold up to B barrier filters associated with each L2 bank.

Each L2 controller can contain multiple barrier filters,

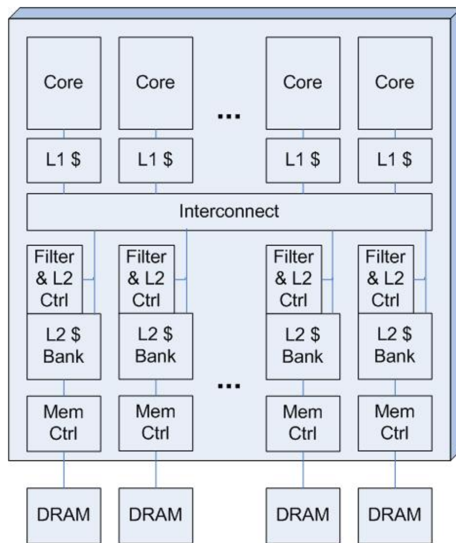

Figure 1. Organization of a standard multi-core augmented with a replicated filter integrated with the L2 cache controller.

each supporting a different barrier being used by a program. Figure 2 shows the state kept track of for a single barrier filter. The barrier filter has an *arrival address tag* and an *exit address tag* and a table containing T entries, where T is the maximum number of threads supported for a barrier. The operating system allocates the cache line addresses for a barrier filter in such a way that the lower bits of the arrival and exit address can be used to distinguish which thread is accessing the barrier filter. This also means that only one arrival address tag needs to be stored to represent all of the arrival addresses, and similarly only one exit address tag needs to be used to identify all of the exit addresses. Therefore, the tags are used to identify the addresses, and the lower bits are used to index into the T entries. Each thread entry contains a valid bit, a *pending fill* bit, indicating for the arrival address if a fill request is currently blocked, and a two bit state machine representing the state of the thread at the barrier. A thread is represented in a barrier filter by its state being in either the Waiting-on-arrival (Waiting), Blocked-until-release (Blocking) or Service-until-exit (Servicing) states. This state is associated with both the arrival and exit address for the thread, which were assigned to this thread by the operating system. Each barrier filter also contains a field called `num-threads`, which is the number of threads participating in the barrier, a counter representing the number of threads that arrived at the barrier (`arrived-counter`), and a last valid entry pointer used when registering threads with the barrier filter.

Each thread will execute the following abstract code sequence to perform the barrier. Some of the steps can be merged or removed depending upon the particulars of the ISA used for a given implementation:

memory fence
invalidate arrival address
discard prefetched data (flush stale copies)

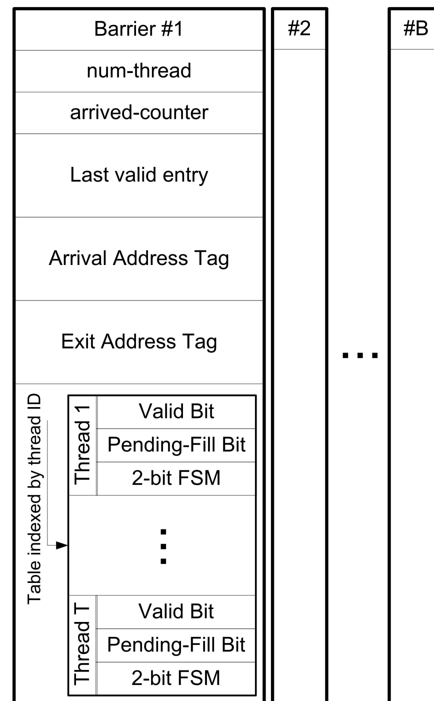

Figure 2. A Barrier Filter Table.

load or execute arrival address
memory fence (needed only for some implementations)
invalidate exit address

Note that all of the above instructions exist on some modern ISAs (e.g., PowerPC), so for those ISAs, no new instructions would have to be added. These architectures contain invalidate instructions as well as memory fence or synchronization instructions.

We assume throughout the rest of this paper that the memory fence ensures that the invalidation of the arrival address only occurs after all prior fetched memory instructions execute. This is achieved, for example, on the Power PC, by executing the `sync` instruction, which guarantees that the following invalidate instruction does not execute before any prior memory instructions. This ordering is enforced to allow the barrier filter to work for out-of-order architectures.

After this code is executed, the thread must tell the barrier filter that it has made it past the barrier. One way to inform the barrier filter is to invalidate the exit address. The exit address is needed because the filter cannot otherwise know that a given thread has received the data from the fill request serviced by the filter. The barrier filter needs to know when each thread has been correctly notified, so that it can then start listening for future arrival invalidations from those threads.

When a barrier is created, it starts off with `arrived-counter` set to zero, `num-threads` set to the number of threads in the barrier, and the Arrival Address and Exit Address tags initialized by the operating system. All of the threads in the barrier start out in the Waiting state as shown in Figure 3.

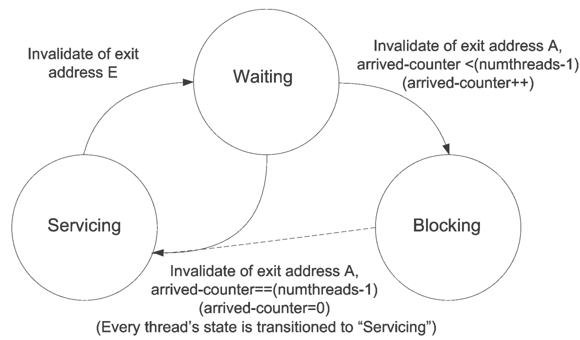

Figure 3. Finite State Automaton that implements the filter for one thread in a given barrier. Both arcs to the Servicing state are described by the lowermost annotation.

The barrier filter then examines invalidate messages that the L2 cache sees. When an address invalidate is seen, an associative lookup is performed in each barrier filter to see if the address matches the arrival or exit address for any of the filters. This lookup is done in parallel with the L2 cache access, and due to the small size of the barrier filter state, the latency of this lookup is much smaller than the L2 access latency. Thus, overall L2 access time would not be adversely affected. In our simulations, we therefore keep L2 latency fixed for both the base and filter variants.

As shown in Figure 3, if the barrier sees the invalidate of an arrival address and the thread's state is Waiting, then the thread's state will transition to the Blocking state, and `arrived-counter` is incremented. If the barrier sees the invalidate of an arrival address and the thread's state is Blocking, then the thread's will stay in the Blocking state.

As shown in the code above, after a thread executes an invalidate for the arrival address, it will then access that address to do a fill request. If a fill request is seen by the barrier filter and the state of the thread is Blocking, then the state will stay blocked, and the pending fill bit for that thread will be set. The fill will not be serviced, because we will only service these fills once all of the threads have accessed the barrier. This pending fill is what blocks the thread's execution waiting for the barrier.

When a thread in state Waiting sees an arrival address invalidation, and arrived-counter is one less than num-threads, we know all of the threads have blocked. We therefore clear the arrived-counter and set all of the states for the threads to Servicing. In addition, we process all of the pending fill requests for the threads. If a fill request comes in for the arrival address and we are in state Servicing, then the fill request is serviced. If the barrier sees the invalidate of an exit address and the thread's state is Servicing, then the thread's state transitions to the Waiting state.

3.2.1 MSHR Utilization

Miss Status Holding Registers (MSHRs) in cores keep track of addresses of outstanding memory references and map them to pending instruction destination registers [9]. Outstanding fill requests to barrier filters thus occupy an MSHR slot in the core originating the request. The MSHR is released once the request is satisfied, or when the thread is context switched out. Hence in a simultaneously multithreaded core, it is important to have at least as many MSHR entries as contexts participating in a barrier. However, providing at least one MSHR entry per SMT context is needed for good performance anyway, so the adoption of barrier filters should not change a core's MSHR storage requirements in practice.

3.3. Interaction with the Operating System

3.3.1 Registering a Barrier

We have assumed that the barrier routines are located within a barrier library provided by the operating system. This library also provides a fall-back software based barrier. We also assume an operating system interface that registers a new barrier with the barrier filter by specifying the number of threads and returns to the user code a barrier handle in response. A request for a new barrier will receive a handle to a filter barrier if one is available, and if there are enough filter entries to support the number of threads requested for the barrier. If the request cannot be satisfied, then the handle returned will be for the fall-back software barrier implementation.

Each thread can then use the OS interface to register itself with the filter using the barrier's handle, receiving the virtual addresses corresponding to the arrival address and the exit address as the response. Once all the threads have registered, then the barrier will be completely constructed and ready to use. Threads entering the barrier before all threads have registered will still stall, as the number of participating threads was determined at the time of barrier creation.

3.3.2 Initializing the Barrier

As discussed above, each thread is assigned a distinct arrival address by the operating system, which maps to a different cache line. Moreover, a given filter must be aware of all addresses assigned to threads participating in the barrier it supports. As pointed out in Section 3.2, the operating system must allocate the cache line addresses for a barrier filter in such a way that the lower bits of the arrival and exit address can be used to distinguish which thread is accessing the barrier filter. With this convention, only one arrival address tag needs to be stored to represent all of the arrival addresses, and similarly only one exit address tag needs to be used to identify all of the exit addresses.

If the target platform has multiple memory channels, as we have been assuming in this paper, memory requests are directed to memory channels depending on their physical target addresses. Care must thus be taken if the filter is distributed across channels, as we again assumed. In this situation, the operating system must make sure that all arrival and exit addresses it provides (for a given barrier) map to the same filter.

The operating system is responsible for initializing filter state, such as the tags for the arrival and exit address, and the counters `arrived-counter` and `num-threads`. To accomplish this, *we assume the data contents of the filters to be memory mapped in kernel mode, obviating a need for additional instructions to manipulate them.*

3.3.3 Context Switch and Swapping Out a Barrier

The operating system can context switch out a stalled thread that is waiting on its fill request, which is blocked by the barrier filter. When this occurs, the fill request for the arrival address will not have committed, so when the thread is rescheduled it will re-issue the fill request again and stall if the barrier has not yet been serviced.

We do not assume pinning to cores, and blocked threads may therefore be rescheduled onto a different core. The distinct arrival and exit addresses for each thread are sufficient to uniquely identify the thread regardless of which core it is running on. If the barrier is still closed when the thread resumes execution, the filter still continues to block this address when the rescheduled thread generates a new fill request. If the barrier opened while the thread was switched out, then when the requesting thread generates a new fill request, the barrier will service that request, as the corresponding exit address has not yet been invalidated. The servicing of the request to the core on which the thread was previously scheduled will not interfere with a subsequent barrier invocation, as any subsequent invocation performs an invalidate prior to again requesting the line on which it may block.

In addition, the operating system can swap out the contents of a barrier filter if it needs to use it for a different application (a different set of threads). When it does this, the operating system will not schedule any threads to execute that are associated with that barrier. Therefore, a barrier represents a co-schedulable group of threads, that are only allowed to be scheduled when their barrier filter is loaded. This also means that the transitive set of threads assigned to a group of barriers needs enough concurrent hardware barrier support to allow these barriers to be loaded into the hardware at the same time. For example, if a Thread X needs to use Barrier A and B, and Thread Y needs to use Barrier B and C, then the OS needs to assign addresses and ensure that there is hardware support so that Barrier A, B and C can be scheduled and used at the same time on the hardware. If not, then an operating system would return an error when a thread tries to add itself to a barrier handle using the previously described interface. The error should then be handled by software.

3.3.4 Implementation Debugging and Incorrect Barrier Usage

Note that the reason why a thread's state does not go directly from Servicing to Blocking is to ensure the operating system's implementation of the barrier is obeying correct semantics, so we can flag these as errors. These few error cases represent invalid transitions in the FSM. If a thread's state in the barrier is in Waiting and a fill request (load) accesses the arrival address then an exception/fault should occur to tell the operating system that it has an incorrect implementation or use of the barrier filter. Similarly, an exception should occur if an invalidate for the arrival address for a thread is seen while the thread is in either the Blocking or Servicing states. Finally, when a thread is in the Waiting or Blocking states and the thread in the barrier filter sees an invalidate for its exit address, an error should also be reported.

The only time that the filter barrier implementations could cause a thread to stall indefinitely is if the barrier is used incorrectly. For example, incorrectly creating a barrier for more threads than are actually being used could cause all of the threads to stall indefinitely. But the same is true for any incorrect usage of any barrier implementation.

Note that unlike a software barrier, which would experience deadlock inside an infinite loop, due to limitations of the constituent cores, a cache miss fill request may not be able to be delayed indefinitely. In this case the filter may generate a reply with an error code embedded in the response to the fill request. Upon receipt of an error code, the error-checking code in the barrier implementation could either retry the barrier or cause an exception. In terms of the barrier implementation itself, Figure 3 can be augmented with exceptions for incorrect state transitions and for error codes embedded in fill responses upon hardware timeouts.

3.4. Detailed Implementations

We examine two specific implementations of our barrier filter. The first uses instruction cache blocks to form the barrier, and the second uses data cache blocks. Both techniques use a similar instruction sequence to that described above.

Note that invalidate instructions only invalidate a single cache line, and for a data cache, the line is written back if dirty on invalidate. These instructions are assumed to be user-mode instructions and to check permissions like any other user mode memory reference. This ensures that that page level permissions are enforced, and that processes only invalidate their own cache lines.

3.4.1 Instruction Cache Barriers

Our I-cache approach leverages a key property of all existing cores, which is that when the next instruction to be executed is fetched and the fetch misses in the I-cache, the thread will stall until the instruction's cache block returns. Here we focus on using code blocks for our arrival and exit addresses, so that when the arrival code block is not being serviced, the thread stalls when it tries to execute code in the block. The instruction sequence is shown below.

memory fence
invalidate arrival address
discard prefetched instructions

execute code at arrival address

Then, after the thread is allowed past the barrier, the thread will perform:

invalidate exit address

For this implementation, the arrival address, which we will denote by A, is assumed to be aligned with the beginning of a single cache line of the first level I-cache. The exit address, which we denote by E, is assumed to be likewise aligned. The size, L, of these lines is no larger than that of the outer cache levels and line inclusion is preserved with respect to outer cache levels.

The arrival sequence contains four instructions: a memory fence, needed both to ensure that all previous operations have been made externally visible before entering the barrier (and via an assumed pipeline flush, to ensure that the next instruction does not execute speculatively); an instruction that invalidates arrival address A; an instruction that discards loaded and prefetched instructions; and an instruction that jumps (or falls through) to the code at address A. Explicit invalidations of the cache line at arrival address A can for example be done using the `ICBI` instruction on the PowerPC architecture. These invalidations are propagated throughout the cache hierarchy above the filter, and they are seen by the barrier filter as described above. The discarding instruction removes any code associated with that block from the current pipeline and any instruction prefetching hardware; this is supported by the instruction `ISYNC` on the PowerPC.

After the code arrival address has been invalidated and the instruction discard executed, the attempted execution of the arrival cache block causes the instruction fetch to stall until the barrier is serviced. Once it is serviced, an invalidate of the exit address would be performed. Note that the exit address could be an instruction or data address. It does not matter, as the content is never accessed.

We assume instruction blocks used for an application's barrier will not be invalidated explicitly except by our barrier mechanism; the line may be evicted from the cache due to replacement, but silently. Prefetching cannot trigger an early opening of the barrier. Data prefetched prior to the invalidate will be invalidated or discarded, and prefetches made after the invalidate are filtered until the barrier opens. The barrier only opens when all threads have explicitly said they have entered the barrier using the invalidate instruction.

3.4.2 Data Cache Barriers

We implement a barrier through the D-Cache by starving loads to arrival addresses until all arrival addresses have been invalidated. Each thread will execute the following code sequence to perform the barrier:

memory fence
invalidate arrival address A
discard prefetches (discard prefetched data from A)
load arrival address A
memory fence

Then, after the thread is allowed past the barrier, the thread will perform:

invalidate exit address E

For this implementation, the invalidate instruction could be done using the `DCBI` instruction on the PowerPC architecture or other equivalent instructions on other architectures that invalidate a specified data cache block. The discard instruction makes sure no prefetched copy of the data is kept internally by the processor. Discarding prefetched data is provided, for example, as a side effect of the store semantics of the PowerPC `DCBI` instruction. The invalidation and discard purges copies of the arrival cache line from cache levels between the core and the filter, making sure the thread will stall on reading the arrival address, as it is not cached, the fill request will be blocked by the barrier filter, and any prefetch buffers potentially containing the requested data have been purged. The arrival sequence finishes with a memory fence instruction, such as the `DSYNC` instruction on the Power PC, or the `mb` instruction on the Alpha, which enforces that no memory instruction is allowed to execute until all prior memory instructions have completed. This means that the load of arrival address A has to complete before any later memory operations can execute. Thus, while additional instructions may continue to execute past the barrier, none that read or write memory may do so, preserving global state, and with it, barrier semantics.

Again, we assume that addresses mapped to data cache lines used for barrier filters for an application will not be invalidated explicitly except to be used for our barrier mechanism; the line may be evicted from the cache, but silently. Once loaded into a barrier filter table, those address ranges should not be used for anything in the application's execution, and only used by the barrier filter library. While a thread is blocked, requests for the arrival address A are stalled; if that cache line is prefetched during this time by hardware, the prefetch will not trigger an early opening of the barrier, since the prefetch will be blocked, because it is a fill request. The barrier only opens when all threads have explicitly invalidated their arrival address using the invalidate instruction.

3.5. Reducing Invalidations Per Invocation: Ping-Pong Versions of Our Barriers

An alternate approach exists for both I-Cache and D-Cache implementations that allows for only a single invalidate to occur per barrier iteration. This is desirable, as invalidations consume non-local bandwidth.

Two barriers are registered, with the arrival address of the first being the exit address of the second, and vice versa. The "exiting" section of a barrier is reduced from an "invalidate and a return" to simply a "return". In a manner somewhat analogous to sense-reversal in classic barrier implementations, which address is invalidated toggles depending on a locally stored variable. As the arrival address for one is the exit for the other, entering the second barrier will exit the first

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

Table 2. Baseline configuration of the multi-core.

and vice versa. Intuitively, when repeatedly executing barriers, a thread ping-pongs between using two logical barriers, giving this barrier version its name.

4. Results

We have been using an unofficial version of SMTSim [23] provided by Jeff Brown and Dean Tullsen at UCSD. We have modified our copy of SMTSim with some additional support for dynamic thread spawning.

SMTSim simulates multi-cores that obey the Alpha architecture, and we used that instruction set with the addition of the PowerPC `ICBI`, `DCBI`, and `ISYNC` instructions. We simulated CMPs with 4, 8, 16, 32, or 64 cores and as many threads, with one thread per core. We focus our examination only on synchronizations occurring among threads executing on a single CMP. We assume all cores are identical. Other simulation parameters are listed in Table 2.

We compare four variants of our barrier filter method (I-Cache, D-Cache, and the ping-pong version of each) with a pure software centralized sense-reversal barrier based on a single counter and single release flag, with a binary combining-tree of such barriers [8], and with a very aggressive implementation of a barrier relying on specialized hardware mechanisms based upon the work of Polychronopolous et al. [3] as previously described in section 2. For the baseline hardware barrier, we assume a two cycle latency to and from the global logic, that the processor will stall immediately after executing the instruction communicating with the global logic, and that the only cost associated with restarting the processor is checking and reseting a local status register.

For our implementation of the sense-reversal software barriers, we use load-linked (`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 [8]. Care was taken to place shared variables (such as the counter and the flag) in separate cache lines to avoid generating useless coherence traffic. The binary combining tree of these barriers features a distinct counter and flag for each pairwise barrier, each on its own cache line.

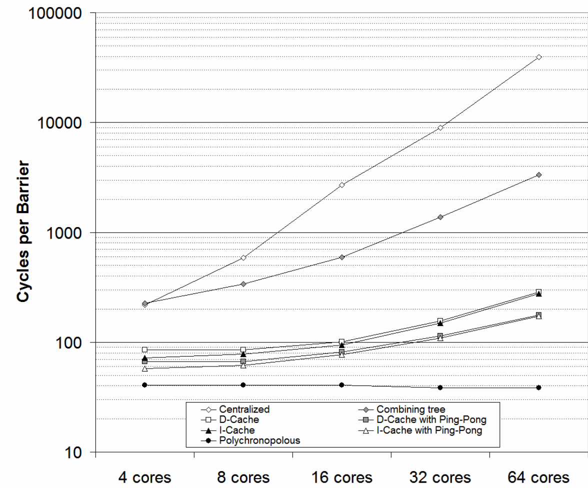

Figure 4. Average execution time of different barrier mechanisms. Lower is better. The top curve corresponds to the software-only centralized barrier.

4.1 Benchmark Selection

To examine the benefit of having fine-grain hardware barriers we looked to the SPLASH-2 suite [25], but we found that the benchmarks there only took advantage of coarse-grain barrier parallelism. The reason for this is that the parallelism was created for multi-chip processors, and not multiprocessors with on-chip communication latencies. For example, the `Ocean` benchmark in our 16-core/16-thread test setup on its default input size (258x258) executes only hundreds of dynamic barriers versus tens of millions of instructions per thread. This leads to barriers accounting for less than 4 percent of total execution time, even with simple, lock-based centralized barriers. While using a filter barrier implementation significantly reduces the overhead from barriers, overall execution only improves by 3.5%.

One focus of this paper is to study the impact of barriers on exploiting fine-grained data parallelism with CMPs, and we could not find any benchmark suites with full fledged applications that used barriers to organize their fine-grained parallelism. We saw tasks often associated with vector processing as fertile ground for fine-grained parallelism that would be exploitable via rapid global (barrier) synchronization. We therefore focused on parallelizing a few of the EEMBC benchmark suite [6] programs along with Livermore loop kernels to take advantage of fine-grain parallelism with barriers.

We have measured the performance of the different barrier mechanisms in two ways. First we measured the latency of the barriers themselves, and then measured the impact of the barrier mechanisms on the performance of various kernels.

4.2 Filter Barrier Latency

Our simulations follow the methodology described in [8]: performance is measured as average time per barrier over a loop of consecutive barriers with no work or delays between them, the loop being executed many times. While this does not model load imbalance between threads, and would therefore be insufficient for examining infrequently executed barriers, it is applicable for barriers associated with a parallelized inner-loop. We constructed loops with 64 consecutive barrier invocations, with the loop being executed 64 times. Our results are shown in Figure 4. Filter-based approaches perform much better than software methods, and scale better as well. However, the scaling of both the filter and software approaches beyond 16 cores was visibly impacted by the saturation of the shared bus resources. The I-cache filter mechanisms have slightly better performance than the D-cache filter mechanisms, in part because they execute only one memory barrier per invocation and the D-cache mechanism must execute two. The sense-reversal versions of each type of filter also perform better than filter barriers with both entrance and exit actions. The sense-reversal variants each perform one invalidation per invocation, while the entry/exit versions perform two, and thus consume greater bus bandwidth. The limited increases in execution time of the barrier filters when going from 4 threads to 16 threads show good scalability in the presence of available bus bandwidth.

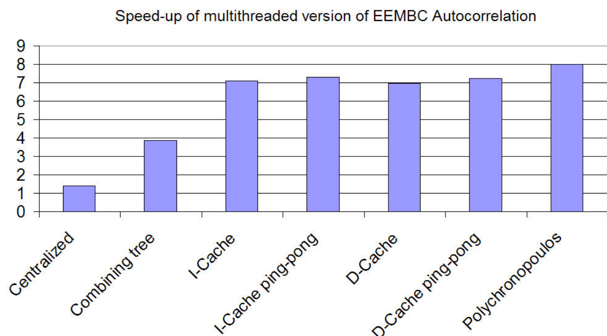

Figure 5. Execution speed-up, relative to sequential execution, of a multi-threaded version of the EEMBC Autocorrelation benchmark, using different barriers.

4.3 Embedded Computing Benchmarks

We looked to embedded benchmarks, such as those focusing on media and telecommunication applications, as likely places to exploit fine-grained parallelism with barriers. We hand-parallelized the Auto-Correlation and Viterbi Decoder kernels from the EEMBC benchmark suite [6], crafting for each a multi-threaded solution based around barriers. The Auto-Correlation kernel is simple, an outer loop that iterates over a lag parameter wrapped around an accumulation loop

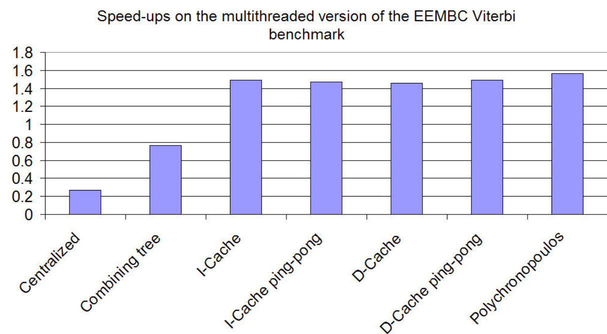

Figure 6. Execution speed-up, relative to sequential execution, of a multi-threaded version of the EEMBC Viterbi benchmark, using different barriers.

dependent upon the input and the lag parameter. We used a pair of barriers to transform the accumulation into a set of parallel accumulations and a reduction. Barriers in the Viterbi Decoder were used to enforce ordering between successive calls to parallelized subroutines.

Figures 4.2 and 4.2 show speedups over sequential execution achieved by the multi-threaded Viterbi Decoder on the `getti.dat` input and by the multi-threaded Auto-Correlation (lag=32) on the `xspeech` input, respectively, when run on 16 cores. The Auto-Correlation benchmark parallelizes readily, with a speedup over sequential execution of 3.86x using software combining barriers, a speedup of 7.31x with the best performing filter barrier, and a speedup of 7.98x using a dedicated barrier network. The Viterbi decoder shows more limited improvements—notably, the parallel implementation using software barriers is actually slower than the sequential version. Only with lower overhead barriers was there a speedup from the multi-threaded approach. Note that in both benchmarks the barrier filter performs almost as well as the aggressively modeled Polychronopoulos barrier hardware, but requires less modification to the cores.

4.4 Livermore Loops

Livermore loops have long been known for being a tough test for compilers and architectures. They present a wide array of challenging kernels where fine-grain parallelism is present but is hard to extract and efficiently exploit. These loop kernels help us illustrate how multi-cores equipped with our mechanisms can be a realistic alternative to vector or special-purpose processors.

We focused on Kernels 2, 3 and 6 of the Livermore suite because the other kernels do not shed better light on the performance and scalability of synchronizations: they are either embarrassingly parallel, such as Kernel 1, or serial, such as Kernels 5 and 20, or similar in structure to another kernel (*e.g.*, Kernels 3 and 4 are both reductions). Loop nest 2 is an excerpt from an incomplete Cholesky conjugate gradient code. A `C` version (transcribed from the original Fortran), as

found on Netlib [17], is shown below.

```
 ii = N;
 ipntp = 0;
 do {
    ipnt = ipntp;
    ipntp += ii;
    ii /= 2;
    i = ipntp;
#pragma nohazard
    for(k=ipnt+1;k<ipntp;k=k+2){
      i++;
      x[i]=x[k]-v[k]*x[k-1]-v[k+1]*x[k+1];
    }
 } while ( ii>1 );
```

Proving that the k-loop has no loop-carried dependence is non-trivial, but the pragma asserts that property. A naive parallelization of the loop would cyclically distribute iterations across cores, generating significant coherence traffic. The version we use partitions arrays in chunks of at least 8 doubles, as that is the size of a cache line. Thus even if the partitions are not aligned with cache lines, cache lines will only need to be transfered between cores at most once. The ID of the current thread is denoted by `MYID`. The value of i, which is the left-hand side subscript, has to be computed from the other variables, as shown in the parallel version of the loop below. Note that the amount of data operated upon, and thus the available parallelism, decreases by a factor of two with successive iterations of the do-while loop.

```
 do{
   ipnt = ipntp;
   ipntp += ii;
   ii /= 2;
   i = ipntp;
   chunk=(ipntp-ipnt)/2+(ipntp-ipnt)%2;
   chunk=chunk/THREADS+((chunk%THREADS)?1:0);
    if (chunk < 8){chunk = 8;}
    i += MYID*chunk;
    end = (chunk*2*(MYID+1))+ipnt+1;
    for( k=ipnt+1+(MYID*2*chunk);
         k<end && k<ipntp; k+=2) {
      ++i;
      x[i]=x[k]-v[k]*x[k-1]-v[k+1]*x[k+1];
    }
    Barrier ();
 } while ( ii>1 );
```

On a CMP with 16 cores each with hardware support for one thread, the performance achieved by various implementations of Loop 2 is shown in Figure 7. In this case the performance of the parallel version using filter barriers does not surpass that of the sequential version until vector lengths of 256 elements are reached (8 elements written per thread by all 16 threads for the most parallel iteration of the do-while loop). The rapid halving of available algorithmic parallelism with each iteration of the do-while loop leads to a slower saturation of available hardware parallelism relative to the other two Livermore loops examined, leading to a qualitatively different curvature over the range of inputs shown.

Loop 3 is a simple inner product, so we don't show its code. Its performance on a CMP with 16 cores, each with hardware support for one thread, is shown in Figure 8. Here the performance of the parallel versions using filter barriers surpasses that of the sequential version at vector lengths as short as 64 elements (8 elements per thread from each input vector, due to the minimum partition size to avoid useless coherence traffic).

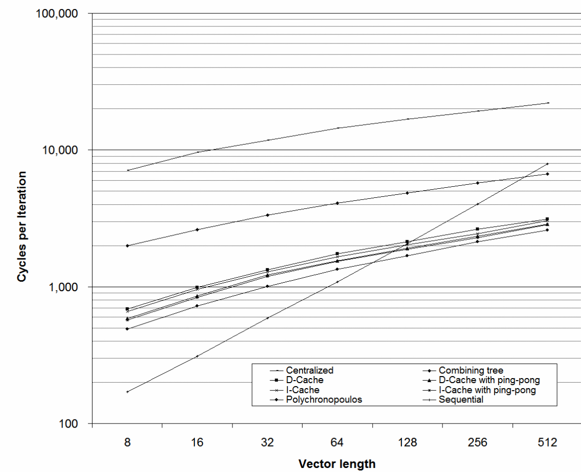

Figure 7. Performance using various barriers on Livermore Loop 2.

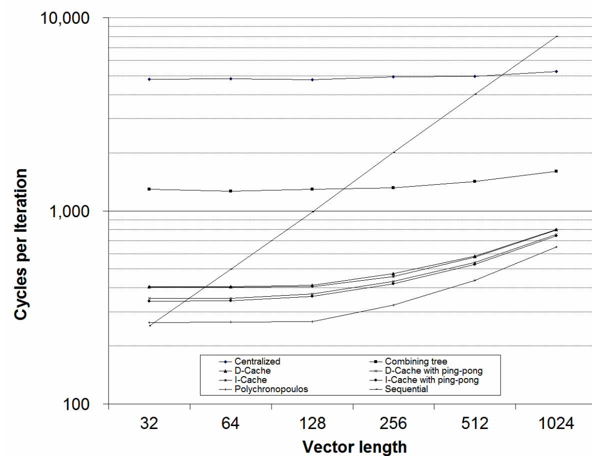

Figure 8. Performance using various barriers on Livermore Loop 3.

Kernel 6 of the Livermore suite is a general linear recurrence equation. The gist of its C code follows:

```
for ( i=1 ; i<N ; i++ ) {
  for ( k=0 ; k<i ; k++ ) {
    w[i] += b[k][i] * w[(i-k)-1];
  }
}
```

To expose parallelism, we invert the k-loop to make k go from i to 0. This results in the data dependences shown as thick arrows in Figure 9, where dots represent instances of the loop body's statements for various values of i and k.

This transformation and the figure make parallel wavefronts clear: instances such that $i-k$ equals 1 only depend on statements before the loop and can therefore be executed first, and simultaneously; then, instances such that $i - k$ equals 2 have their incoming dependencies resolved and can execute in parallel. This process can be pictured as a new coordinate axis, t, which indicates the time step at which an instance can be performed.

This representation naturally yields to our multi-threaded code, which is constructed as follows. In theory, k could simply equal `MYID`. However, since we want to assess the achieved performance on CMPs of up to 16 cores with support for as few as one thread per core, our code explicitly handles multiple ks per thread.

```
for (t=0; t<=N-2; t++) {
  for(k=MYID*CHUNK;k<(MYID+1)*CHUNK;k++) {
    if(k<(N-t)){w[t+k+1]+=b[k][t+k+1]*w[t];}
  }
  Barrier();
}
```

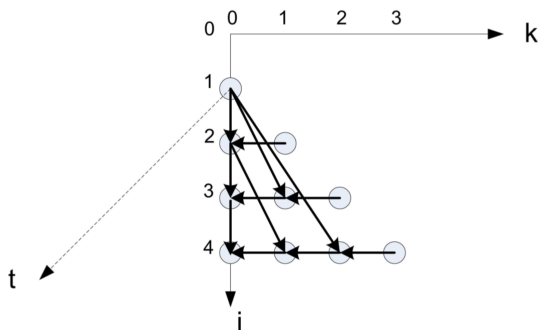

Figure 9. Original and transformed iteration space of Livermore Loop 6.

This code and Figure 9 make clear that this example is not embarrassingly parallel and has a decreasing amount of parallelism. Also, the parallelism is very fine grained and could not be efficiently exploited on a CMP without fast synchronization. In addition, the required synchronizations have an irregular pattern that doesn't make them amenable to point-to-point synchronizations. Therefore, a global barrier synchronization is a natural choice in this code.

Figure 10 shows the execution time of the sequential and multi-threaded versions on an 16-core CMP, each with one thread, for different vector (input `w`) sizes N (and thus input `b` sizes NxN) and different barrier implementations. Using these techniques, the fast barrier synchronization provided by barrier filters allows the 16-thread version of the parallel code to be faster than a sequential version (which, of course, has no synchronization overhead) at vector lengths as small as

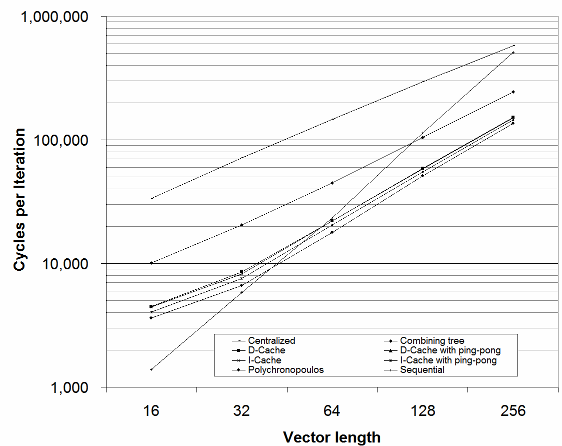

Figure 10. Performance using various barriers on Livermore Loop 6.

64 elements. The parallel version is more than a factor of 3 faster exploiting the fine-grain inner loop parallelism than the sequential version for vector lengths of 256 elements.

5. Summary

We have shown that a CMP can be used to exploit very fine-grained data parallelism, expanding the range of code structures subject to speedup through multi-threading to include those traditionally accelerated with vector processing (i.e., inner loop parallelism). Straightforward parallelizations of such code, however, require frequent execution of barriers, making overall performance sensitive to barrier implementation. We show that fast barriers, with hardware support that capitalizes upon the low, on-chip latencies between cores on a CMP, can significantly improve performance when using a CMP to exploit fine-grained data parallelism.

We have presented a mechanism for barrier synchronization that does not rely on locks, nor busy waiting on a shared variable in memory, and generates no spurious coherence traffic. Instead, it leverages a simple key idea: we make sure threads at a barrier require an unavailable cache line to proceed, and we starve their requests until they all have arrived. The implementation relies on additional logic that blocks specific cache fills at synchronization points. No coherence traffic for exclusive ownership is required for instruction cache fills or data cache read miss fills. This is in contrast to software barrier methods which update shared barrier state variables.

We evaluated the performance of barrier filters using a number of techniques. We evaluated the performance of various types of barriers in isolation. Both our instruction and data filter barriers were competitive with aggressive implementations of previously proposed hardware synchronization networks, which require modification of the processor core, up until bus bandwith was saturated. The instruction cache

fill barriers were somewhat faster than data cache fill barriers, and the ping-pong variants were likewise faster than their corresponding arrival/exit implementations, as the reduction in invalidations reduced consumption of bus bandwidth.

We also evaluated the impact of fast barrier synchronization on a number of kernels. On the `Auto-Correlation` and `Viterbi Decoder` benchmarks from the EEMBC suite, multi-threaded implementations using barrier filters had speedups around twice that of software tree barriers and almost as good as a dedicated barrier network but without modifying the cores. Finally, we showed that fast barrier synchronization enabled speedup on Livermore Loop kernels with modest vector lengths, whereas software-only barriers required vector lengths longer by a factor of two to four to achieve a speedup. Based on these results, we believe fast barrier synchronization using barrier filters could pave the way for efficiently exploiting fine-grained parallelism on CMPs utilizing minimally modified cores.

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] B. Beck, B. Kasten, and S. Thakkar. Vlsi assist for a multiprocessor. In *Proceedings of the second international conference on Architectural Support for Programming Languages and Operating Systems*, pages 10–20, 1987.

[3] C. J. Beckmann 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] L. Cheng and J. Carter. Fast barriers for scalable ccNUMA systems. In *International Conference on Parallel Processing*, pages 241–250, 2005.

[6] E. M. B. Consortium. `www.eembc.org`.

[7] P. Coteus et al. Packaging the Blue Gene/L supercomputer. *IBM Journal of Research and Development*, 49(2/3):213–248, Mar. 2005.

[8] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture*. Morgan Kaufmann, 1999.

[9] K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *Proc. Intl. Symp. on Computer Arch. (ISCA)*, pages 211–222, 1994.

[10] J. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent shared-memory multiprocessors. In *Proc. of 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1989.

[11] A. Gottlieb et al. The NYU ultracomputer – designing a MIMD, shared-memory parallel machine. In *Proceedings of the 9th annual symposium on Computer Architecture*, pages 27–42, 1982.

[12] W. T.-Y. Hsu and P.-C. Yew. An effective synchronization network for hot-spot accesses. *ACM Transactions on Computer Systems (TOCS)*, 10(3), Aug. 1992.

[13] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 chip: a dual-core multithreaded processor. *IEEE Micro*, pages 40–47, March-April 2004.

[14] S. W. Keckler et al. Exploiting fine-grain thread level parallelism on the mit multi-alu processor. In *Proceedings of the 25th annual International Symposium on Computer Architecture*, pages 306–317, 1998.

[15] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, Mar. 2005.

[16] C. E. Leiserson et al. The network architecture of the Connection Machine CM-5. In *Proc. of SPAA*, pages 272–285, June 1992.

[17] Livermore loops coded in C. *http://www.netlib.org/benchmark/livermorec*.

[18] 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.

[19] D. S. Nikolopolous and T. S. Papatheodorou. Fast synchronization on scalable cache-coherent multiprocessors using hybrid primitives. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, 2000.

[20] 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.

[21] S. L. Scott. Synchronization and communication in the t3e multiprocessor. In *Proc. of 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1996.

[22] S. Shang and K. Hwang. Distributed hardwired barrier synchronization for scalable multiprocessor clusters. *ACM Transactions on Parallel and Distributed Systems (TPDS)*, 6(6), 1995.

[23] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, December 1996.

[24] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fine-grained synchronization on a simulataneous multithreading processor. In *Proc. Int'l Symp on High-Performance Architecture (HPCA)*, Jan. 1999.

[25] S. C. Woo et al. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. 22nd Intl, Symp. on Computer Arch.*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[26] D. Yeung and A. Agarwal. Experience with fine-grain synchronization in mimd machines for preconditioned conjugate gradient. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 187–192, 1993.