Performance of Switch Blocking on Multithreaded Architectures

K. Gopinath & Krishna Narasimhan M.K.
Department of Computer Science & Automation
Indian Institute of Science, Bangalore
and
B.H. Lim & Anant Agarwal
Laboratory for Computer Science
MIT, Cambridge, US

September 1, 2010

Abstract Block multithreaded architectures tolerate large memory and synchronization latencies by switching contexts on every remote-memory-access or on a failed synchronization request. The Alewife machine, developed at the MIT, is one such architecture. We study the performance of a waiting mechanism called switch-blocking on Alewife and compare it with switch-spinning.

1 Introduction

Remote memory access latencies are increasingly becoming high in large-scale multiprocessors[Dash]. They cannot be removed entirely by caching as some memory transactions can cause cache coherence protocols to issue invalidation messages and misses have to be endured. Processors can spend most of their time waiting for remote accesses to be serviced, hence reducing the processor utilization[Dash91]. A similar problem arises when the processor must wait due to a synchronization; the problem is even more acute as these delays may be unbounded.

One solution that addresses *both* the above mentioned problems allows the processor to have multiple outstanding remote memory accesses or synchronization requests. The Alewife [CHA91] machine implements this solution by using a processor that can switch rapidly between multiple threads of computation and a cache controller that supports multiple outstanding requests. Processors that switch rapidly between multiple threads of computation are called *multithreaded architectures.*

The prototypical multithreaded machine is the **HEP** [SMI81]. In **HEP**, the processor switches every cycle between eight processor resident threads. Architectures employing such cycle-by-cycle interleaving of threads are called *finely multithreaded.* In contrast, Alewife employs *block multithreading*: context switches occur only when a thread executes a memory request that must be serviced by a remote node in the multiprocessor, or on a failed synchronization request. The Alewife controller traps the processor and the trap handler forces a context switch. The trap handling routine has a variety of waiting mechanisms to choose: spinning, switch-spinning, blocking and switch-blocking.

Previous work at MIT[LIM91] had studied the performance of the first three waiting mechanisms under various models (2-phase algorithms, matched and unmatched algorithms, etc. described below). Here we report the modelling, implementation and benchmark performance of switch-blocking. We also compare switch-blocking with *switch-spinning* and draw some conclusions.

This paper is organized as follows. In Section 2, we describe briefly Alewife's architecture, synchronization constructs and waiting mechanisms. In Section 3.2, we propose a Markov model for switch-blocking and compute its effectiveness. Section 4 gives the implementation details of switch-blocking. Section 5 presents the software simulator (ASIM) & benchmarks used and the results, especially comparing one-phase switch-spinning with switch-blocking and the two-phase switch/block algorithms for these two waiting mechanisms by running different benchmarks on Alewife. Finally, we conclude with some observations and future work.

2 Alewife Machine

Before we proceed, it is judicious to define some terms. A *thread* is a process with its own processor state but without a separate address space. A parallel program consists of a set of inter-communicating threads. A *hardware context* is a set of registers on a processor that implements the processor-resident state of a thread. A multithreaded processor has several hardware contexts; Alewife provides four on each processor. A *context switch* is a transfer of processor control from one processor-resident thread to another processor-resident thread. No thread state needs to be saved out into memory. *Loading* a thread refers to the action of installing the state of a thread into a hardware context on a processor and *unloading* a thread refers to the complimentary action of saving the processor-resident state of a thread into memory. A thread is *blocked* when its execution is suspended and it is queued on some synchronization condition. The thread remains blocked until it is signaled to proceed by another thread.

Alewife is a SPARC-based cache-coherent, block-multithreaded, distributed memory multiprocessor that supports a shared-memory programming abstraction. The high-level hardware organization of an Alewife node (Figure **??**) consists of a processor, 64K bytes of direct-mapped cache, 4M bytes of globally-shared main memory, cache-memory-network controller, a network switch , and a floating-point co-processor. Each node has an additional 4M bytes of local memory, a portion of which is used for the coherence directory. The cache-memory-network controller is responsible for synthesizing a globally shared address space from the distributed memory nodes. The 32-bit address space on SPARC limits the maximum machine size to 512 nodes. Alewife has a simple memory mapping scheme, the top few bits of the address determine the node number, and the rest of the address is the index within the specific node.

The initial implementation of the node processor is variously called Sparcle or APRIL [KRA90] and designed to meet several requirements that are crucial for multiprocessing: tolerate latencies and handle traps efficiently. It has four hardware contexts with a context switch taking 14 cycles. The trap mechanism takes 5 cycles to empty the processor pipeline and save the relevant state before passing control to the trap handler.

Alewife implements the directory-based LimitLESS[1] cache coherence scheme [KUB91], [CHA90] that implements a small set of pointers in memory, as do limited-directory protocols. But, when necessary, the scheme allows a memory module to interrupt its local processor for the software emulation of a full-map directory: this is facilitated by Sparcle's fast trap mechanism.

2.1 Mapping of Threads onto Hardware Contexts

Figure **??** illustrates how threads are mapped onto the hardware contexts of the processor. The left half of the figure depicts the user-visible state comprising four sets of general purpose registers and four sets of Program Counter(PC) chains and Processor Status Registers(PSR). The PC chain represents the instruction addresses corresponding to a thread, and the PSR holds various pieces of processor-specific state. Each register set, together with a single PC-chain and PSR, constitutes a hardware context. Only one hardware context is active at a given time and is designated by a current frame pointer(FP). All register accesses are made to the active register set and instructions are fetched using the active PC-chain. Additionally, a set of 8 global registers are provided, these are accessible regardless of FP. In the Figure **??**, $0x$ denotes context 0, $1x$ denotes context 1, etc.

It is important to emphasize the difference between traditional view of context switching and thread context switching in Alewife. Traditionally, a context switch involves saving out the state of a process into memory, and loading the state of another process into the processor. In multithreaded architectures, a context switch does not involve saving state into memory; the processor can activate a different hardware context. Processor state is saved in memory only when a thread is blocked and unloaded. The time in between context switches is termed as *context run length.*

A thread on Alewife can be in any one of four states: *new, running, blocked* and *re-enabled.* A thread is *running* if it is resident in a hardware context regardless of whether it has control of the processor. Blocking a thread involves unloading a thread and placing it on a queue associated with the failed synchronization. Re-enabling a blocked thread involves removing the thread from the blocked queue and queuing that thread on the processor on which it was originally running.

Associated with each processor are three logical queues. One queue is used for new threads, a second queue is used for re-enabled threads, and a third queue is used for lazy task creation [MOH90]. A distributed scheduler is responsible for finding work on these queues. By default, each processor that is not fully loaded runs the scheduler in one of the free contexts. The scheduler forages for work by probing its logical queues and the queues of remote processors, favoring physically closer processors over farther processors in order to enhance communication locality.

2.2 Synchronization in Alewife

Alewife's primitives for hardware support for synchronization are full/empty bits and an efficient trap mechanism. Higher-level synchronization constructs are synthesized in software using these relatively minimal hardware primitives and are available as a software library. The trap mechanism is used to efficiently detect synchronization faults and relegate control to a trap handler to execute an appropriate waiting algorithm.

A full/empty bit, like in HEP [SMI81], is associated with each memory word and atomic **read/modify/write** operations can be performed on the full/empty bit in a single cycle. The **read-and-empty** is an atomic instruction that simultaneously reads a memory word and sets the associated full/empty bit to empty. An *empty-location* trap is generated if the word was already tagged as empty. Conversely, **write-and-fill** is another atomic instruction that simultaneously writes a memory word and sets the associated full/empty bit to full. A *full-location* trap is generated if the word is already tagged as full. A synchronization *succeeds* if it does not cause a thread to wait, and *fails* otherwise.

The following higher-level synchronization constructs provided by Alewife are built on top of the hardware full/empty bit: semaphores, mutual-exclusion locks, read/write locks, J-structures (reusable I-structures), L-structures, futures and barriers. We discuss briefly only those that are unique to Alewife below.

J-structures (reusable I-structures) are implemented as vectors with full/empty bits associated with each vector slot. A reader of a J-structure slot is forced to wait if the full/empty bit for that slot is 0. A writer of a J-structure slot sets the full/empty bit to 1 and releases any readers waiting on it. A J-structure can be *reset.* This clears out all the value slots and sets the full/empty bits to 0. Resetting a J-structure to an empty state enables efficient memory allocation and good cache performance.

[1] *Limited* directory *Locally Extended through Software Support*

L-structures are similar to J-structures but support three operations: locking read, non-locking read and synchronous write. A locking read waits until an element is full before emptying it (i.e. locking it) and returning the value. A non-locking read also waits until the element is full, but then returns the value without emptying the element. A synchronizing write stores a value to an empty element, and sets it to full, releasing any waiters. An L-structure therefore allows mutually exclusive access to each of its elements. In addition, L-structures allow multiple non-locking readers.

Futures [HAL90] are a way of specifying parallelism in Multilisp. They encapsulate both a partitioning of a program into threads and synchronization on the return values of the threads. The evaluation of **(future X)**, where X is an arbitrary expression, creates a thread to evaluate X and also creates an object known as a *future* to eventually hold the value of X. When created, the future is in an *unresolved* state. When the value of X becomes known , the future *resolves* to that value. Concurrency arises because the expression **(future X)** returns the future as its value without waiting for the future to resolve. Thus, the computation containing **(future X)** can proceed concurrently with the evaluation of X.

The future object is a first-class object whose type is checked at run-time. Without some sort of hardware support, the check would have to be done in software, incurring substantial cost. However, creative use of SPARC non-fixnum traps and compiler enforced addressing conventions allow the detection of futures as efficiently as having hardware tag checking without additional hardware.

It is implemented as two memory words: one to eventually hold the value of the future and another to hold the queue of blocked waiters(see Figure **??**). This data structure is identical to the data structure for semaphores. However the data structure is used in a very different way. Any pointer to a future object has its least significant three bits set to 101 to support automatic type checking. Also, the value slot for the future starts out in the empty state with a pointer to the thread closure associated with the future. The value slot is written to at most once, when the future is resolved.

2.3 Waiting Mechanisms

Spinning is a polling mechanism. It has low execution cost because each poll of the synchronization variable consumes only a few processor cycles to read a memory location, but it is not processor-efficient because it prevents the other threads from utilizing the processor.

When a thread spin-waits, it periodically monitors the state of a memory location. Besides consuming processor cycles, it can also cause the load on the network to increase if not done carefully. An economical way to spin-wait on a machine with hardware cache coherence like Alewife is to continuously read a memory location. In this way, the location gets cached locally, and no network bandwidth is consumed while spinning. When the state of the memory location changes due to a write, the ensuing cache invalidations cause the spinning waiters to notice the change and attempt to synchronize again.

Blocking is a signaling mechanism. It is processor-efficient because it relinquishes control of the processor, but has high execution cost. This cost includes the time needed to *unload* and suspend the waiting thread, and then reschedule and *reload* it at a later time. Unloading a thread involves storing its hardware context into memory and reloading a thread involves restoring the saved context of the thread onto the processor.

Alewife does not provide any special hardware support for unloading. When a thread is blocked, its user registers and state registers are first written out to memory. The suspended thread is then placed on a queue associated with the synchronization location being waited on. When another thread updates the synchronization location, it re-enables all waiting threads by queuing them on their respective processor queues. A thread is reactivated by reloading its saved registers into the processor, and then picking up where it left off.

In the case of *switch-spinning*, context switch takes place to another resident thread upon a synchronization fault. The processor is kept busy executing other threads at the cost of the fast context switch without incurring the high overhead of blocking. Control eventually returns to the waiting thread and the failed synchronization is retried. This mechanism is more processor-efficient than spinning and has a lower execution cost than blocking.

The *switch-blocking* waiting mechanism *disables* the context associated with the waiting thread and switches execution to another context. The disabled context is ignored by further contexts switches until it is re-enabled when the waiting thread is allowed to proceed. Contrast this with blocking, which requires unloading a thread. It is as processor-efficient as blocking and has a low-execution cost because threads are not unloaded.

3 Two Phase Algorithms

Without any information about the distribution of wait times, one cannot expect an on-line waiting algorithm to perform as well as an optimal optimal off-line algorithm. However, competitive analysis can be used to place an upper bound on the cost of an on-line algorithm relative to the cost of an optimal off-line algorithm. A *c-competitive* algorithm has a cost that is at most *c* times more than the cost of an optimal off-line algorithm. Karlin[KAR90] present a refinement of the 2-phase blocking scheme, and prove a competitive factor of 1.59 on their algorithm. The idea is based on 2-phase blocking: given a choice between spinning and blocking, the waiting thread should spin for some period of time and then block if the thread is still not allowed to proceed. The maximum lenngth of the spin phase is randomly picked from a predetermined probability distribution.

An optimal off-line algorithm has perfect knowledge about wait times. Therefore, at each wait, it knows exactly how long the wait time will be. If the cost of switch-blocking for the entire wait time is more than the cost of blocking, the optimal off-line algorithm will choose to block immediately. Otherwise it will switch-block. A two-phase algorithm splits the waiting into a polling phase and a signaling phase. But we will be considering a two-phase algorithm that splits the waiting between two signaling phases. This algorithm executes switch-blocking for some length of time and then blocking if the thread still has to wait.

Let us express the length of the switch-blocking phase as αB, where α is a non-negative constant and B is the overhead of blocking. Setting α to 1 yields a 2-competitive algorithm because any wait time that is less than B incurs the same cost as an optimal off-line algorithm while any wait time that is more than B cycles is exactly twice the cost of the optimal algorithm. If $\alpha > 1$, any wait time that is more than B cycles costs $(1+\alpha)$ times more than optimal.

The following abbreviations are used for naming the waiting algorithms:

S_{sbl}	-	always switch-block
block	-	always block
$OptS_{sblB}$	-	optimal two-phase off-line using switch-blocking and blocking
$S_{sblB\alpha}$	-	two-phase switch-block/block

Here S_{sblB} denotes that the waiting thread will switch-block for some amount of time and then it will block itself, if the condition waited upon is not satisfied.

3.1 Expected Costs

In the analysis that follows, we write random variables in upper-case and their values in lower-case. The probability density function (PDF) of the random variable X is denoted by $f(x)$. Let t be the average wait time. Let T denote the random variable for wait time and the PDF for wait time be denoted by $f(t)$. Let B be the cost of blocking.

Always Switch-Block (S_{sbl}) If the cost of switch-blocking is $\frac{t}{\gamma}$, then $\frac{1}{\gamma}$ will denote the fraction of processor time wasted on a synchronization condition. From the model that we have proposed for switch-blocking we can derive the equation for γ as follows

$$\gamma = \frac{t}{W} \tag{1}$$

Cost of waiting for t cycles under S_{sbl} is t/γ, the expected cost of S_{sbl} is

$$\begin{aligned} E[C_{S_{sbl}}] &= \frac{1}{\gamma}\int_0^\infty tf(t)dt \\ &= \frac{1}{\gamma}E[T] \end{aligned} \tag{2}$$

Blocking (*block*) the cost of blocking is B, regardless of the wait time distribution.

$$E[C_{block}] = B \tag{3}$$

Optimal switch-block/block ($OptS_{sblB}$) The cost of doing switch-blocking in phase 1 will reduce the cost of that phase by a factor of γ and the maximum length of this phase will be γB. Hence

$$E[C_{OptS_{sblB}}] = \frac{1}{\gamma}\int_0^{\gamma B} tf(t)dt + B\int_{\gamma B}^\infty f(t)dt \tag{4}$$

Two-phase Switch-Block/Block ($S_{sblB\alpha}$) This algorithm switch-blocks until the cost of switch-blocking is equal to αB and then blocks if necessary. Hence the expected cost of $S_{sblB\alpha}$ is

$$E[C_{S_{sblB\alpha}}] = \frac{1}{\gamma}\int_0^{\alpha\gamma B} tf(t)dt + (1+\alpha)B\int_{\alpha\gamma B}^\infty f(t)dt \tag{5}$$

Once the wait time distributions for commonly used synchronization types (like mutual exclusion, barriers, producer-consumer) are derived by making suitable assumptions (see Lim[LIM91] for details), one can compute the cost of the different waiting algorithms. Given that t is the synchronization wait time, Lim[LIM91] assumed that it is t/β for switch-spinning, where β is given by

$$\beta = \frac{t}{(x+C)\lceil\frac{t}{N(x+C)}\rceil} \tag{6}$$

Here $\lceil\frac{t}{N(x+C)}\rceil$ denotes the number of times control returns to the waiting thread before it succeeds in its synchronization attempt and x denotes the context run-length. Assuming that the factor for switch-blocking is γ instead of β, all the results that Lim derives are carried through except that β has been replaced by γ. For completeness, we include only the results for producer-consumer.

3.1.1 Waiting Cost for Producer-Consumer

Assume a single-producer, multiple-consumer situation to model the synchronization provided by futures and J-structures. Let us assume a Poisson process with arrival rate λ. The PDF of wait times for consumers is the exponential distribution

$$f(t) = \lambda e^{-\lambda t} \tag{7}$$

Hence the cost of switch-block is

$$E[C_{S_{sbl}}] = \int_0^\infty \frac{t}{\gamma}\lambda e^{-\lambda t}dt = \frac{1}{\lambda\gamma} \tag{8}$$

The cost of optimal off-line switch-block/block is

$$\begin{aligned} E[C_{OptS_{sblB}}] &= \int_0^{\gamma B}\frac{t}{\gamma}\lambda e^{-\lambda t}dt + B\int_{\gamma B}^\infty \lambda e^{-\lambda t}dt \\ &= \frac{1}{\lambda\gamma}(1-e^{-\lambda\gamma B}) \end{aligned} \tag{9}$$

The cost of two-phase switch-block/block is

$$\begin{aligned} E[C_{S_{sblB\alpha}}] &= \int_0^{\gamma\alpha B}\frac{t}{\gamma}\lambda e^{-\lambda t}dt + (1+\alpha)B\int_{\gamma\alpha B}^\infty \lambda e^{-\lambda t}dt \\ &= \frac{1}{\lambda\gamma}(1-e^{-\lambda\alpha\gamma B}) + Be^{-\lambda\alpha\gamma B} \end{aligned} \tag{10}$$

3.2 Markov Model for Switch-Blocking

The above derivations do not take into account the cost of finite context switching times in hardware, though small compared to B. To incorporate this into the model, we develop the following Markov model for pure switch-spinning and switch-blocking for the matched case where the number of threads and processor contexts match. At the level of modelling attempted, it is not possible to derive the differences between them in one stroke; we use the sojourn times computed below in the combined model to then estimate the additional overhead of switch-spinning over switch-blocking. (We leave the modelling for the unmatched case as future work.)

3.3 Combined Model

We make the following assumptions:

- The average rate at which any active context gets disabled is exponentially distributed with parameter λ. Note that this is not the same as the rate at which a particular context gets disabled as the exact identity of the context is not clear in the case of switch-blocking.
- The average rate at which a context is enabled is exponentially distributed with parameter μ.
- At any point in time only one context can be enabled or disabled.

Let N be the total number of hardware contexts. Let π_i denote the steady state probability of state i. Let C denote the context switch overhead. Let $E[T_b]$ denote the expected time when all the contexts are disabled. Let $E[T_{bi}]$ denote the expected time when i contexts are disabled. Let $E[T_{wi}]$ denote the expected wait time of context i.

The Markov model has $N+1$ states. State i denotes the state wherein i contexts are enabled. State 0 denotes the state wherein all the contexts are disabled. Initially all the contexts are enabled. The rate at which a transition takes place from state $(N-i)$ to state $(N-i+1)$ is the rate at which any one of the remaining i contexts are enabled. Hence this rate is $i\mu$.

The rate at which a transition takes place from state $(N-i)$ to state $(N-i-1)$ is the minimum rate at which any one of the remaining $(N-i)$ contexts are disabled. Hence this rate is $(N-i)\lambda$. Figure **??** illustrates this model. The steady state probabilities can be calculated as follows:

$$\pi_N N\lambda = \mu\pi_{N-1}$$

$$\pi_{(N-2)}2\mu = (N-1)\lambda\pi_{N-1}\vdots$$

Hence we get

$$\pi_{(N-i)} = \frac{N!}{i!(N-i)!}(\frac{\lambda}{\mu})^i\pi_N \quad for\ i = 1,\ldots,N.$$

Since $\sum_{i=0}^{N}\pi_i = 1$, we get

$$\pi_N = \frac{1}{\sum_{i=0}^{N}\frac{N!}{i!(N-i)!}(\frac{\lambda}{\mu})^i}$$

The rate at which the state $(N-i)$ transits to state $(N-i+1)$ is exponentially distributed with parameter $i\mu$ and the rate at which a transition takes place to state $(N-i-1)$ is also exponentially distributed with parameter $(N-i)\lambda$. Hence the time spent in the state $(N-i)$ is again exponentially distributed with parameter $min((N-i)\lambda, i\mu)$, which is same as $(N-i)\lambda + i\mu$: this leads to equation 11

$$E[T_{bi}] = \frac{1}{(N-i)\lambda + i\mu} \tag{11}$$

Substituting $i = N$ in equation 11 we get

$$E[T_b] = \frac{1}{N\mu} \tag{12}$$

The PMF's for different values of λ/μ are plotted in Figures **??** and **??**.

3.4 Modelling the difference

To model the difference between switch-spinning and switch-blocking, we make the assumption that the sojourn times for the various times are the same for both and given by the above equations. This approximation is reasonable as they difference in the simulated times for many benchmarks is typically not more than 5-8%.

The following result for M/G/1 with vacations can be used immediately[NET87] where W is the wait time, X is the service time, V the vacation:

$$W = \lambda\overline{X^2}/2(1-\rho) + \overline{V^2}/2\overline{V} \tag{13}$$

Let the service time include the context switch time. Then the vacation time is given by kC_{sp} where k is the number of idle contexts attempted before success in switch-spinning, C_{sp} the context switch time for switch-spinning and C_{sb} for switch-blocking. ($C_{sp} = 14$ and

$$C_{sb} = C_{sp} + 2 = 16$$

: see the section on implementation)

The vacation part of the waiting time in equation 13 is computed as follows:

$$\overline{V_{sp}} = C_{sp}(1-\pi_0)(\pi_4{*}0+\pi_3/3+\pi_2(1/3+2/3)+3\pi_1)+\pi_0 E[T_b] = xC_{sp}(1- \tag{14}$$

$$\overline{V_{sp}^2} = C_{sp}^2(1-\pi_0)(\pi_4{*}0+\pi_3/9+\pi_2+9\pi_1)+\pi_0 E[T_b]^2 = yC_{sp}^2(1-\pi_0)+\pi_0 E \tag{15}$$

$$\overline{V_{sb}} = (1-\pi_0)(C_{sb}-C_{sp}) + \pi_0 E[T_b] \tag{16}$$

$$\overline{V_{sb}^2} = (1-\pi_0)(C_{sb}-C_{sp})^2 + \pi_0 E[T_b]^2$$

(17)

Given that the number of enabled contexts is i, and the current context is enabled, the average is computed by enumerating all the possible states of the contexts, and, assuming that they are equiprobable, multiplying by the time for the number of intervening failed contexts. Switch-blocking has a short vacation with each context-switch given by $(C_{sb} - C_{sp})$.

To estimate how effective the model is, consider the case for $\lambda/\mu = 0.5$ that is close to the observed probablity profile for the number of live contexts for the multigrid benchmark. The computed values are as follows: $\pi_0 = 0.012; \pi_1 = 0.099; \pi_2 = 0.296; \pi_3 = 0.395; \pi_4 = 0.198$

Using 12 and assuming an average value of context run length obtained from a simulation for multigrid (namely: 43, i.e. $\lambda = 1/43$ per cycle), the extra wait for switch-spinning is given by $0.69C_{sp}$ i.e. for every context-switch suffered by switch-blocking, there is an extra 0.69 context-switch overhead in the case of switch-spinning. This roughly corresponds to the observed difference in clock cycles between switch-blocking and switch-spinning in order of magnitude (calculated: 24,000 cycles; observed: 33,000 cycles). This is quite reasonable as we have not explicitly taken into account the context switches due to synchronization or remote cache misses. More work is in progress to validate the model comprehensively.

4 Implementation of Switch Blocking

The Sparcle processor is based on the following modifications to SPARC architecture:

- Register windows in the SPARC processor permit a simple implementation of block multithreading. A window is allocated to each thread. The current register window is altered via SPARC instructions (**SAVE** and **RESTORE**). To effect a context switch, the trap routine saves the Program Counter (PC) and Processor Status Register (PSR), flushes the pipeline and sets the Current Window Pointer (CWP) to a new register window for a total of 14 cycles.
- The emulation of multiple hardware contexts in the SPARC floating-point unit is achieved by modifying floating-point instructions in a context-dependent fashion as they are loaded into the FPU and by maintaining four different sets of condition bits.
- Sparcle detects unresolved **future** through SPARC *word-alignment* and tagged-arithmetic traps, with the non-fixnum trap modified to look at only the low bit.
- Through the use of *memory exception* (MEXC) line on SPARC, the controller can invoke synchronous traps and rapid context switching.
- SPARC architecture definition includes an *alternate space indicator* (ASI) feature that permits a simple implementation of an interface with the controller. The ASI is externally available as an eight-bit field and is set using SPARC's load and store instructions (**LDA** and **STA**). By examining the processors ASI bits during memory accesses,the controller can select between different load/store and synchronization behavior.

SPARC provides two registers called **WIM** and **CSR**. The *window invalid mask* (**WIM**) is a 32 bit register. The context-specific bits in the mask indicate whether the context is enabled or disabled. A 0 in the bit corresponding to a context indicates that the context is disabled, whereas a 1 indicates that the context is enabled.

Since each context is allocated two registers, the **WIM** allocates two bits to each context. For e.g. (11111100) denotes context 0, (11110011) denotes context 1, etc. Since Alewife has only 4 hardware contexts, only the last 8 bits were considered for numbering the contexts, the other bits being zero.

CSR (*context status register*) is a 32 bit register in which the top 4 bits are zeros, bits 24 through 27 store the number of free contexts, bits 20 through 23 store the total number of contexts and bits 0 through 19 store a mask called *context empty mask* (**CEM**). The simulator was modified to include these registers. Note that the number of free contexts is equal to number of 1's in the **CEM** bit-vector.

Sparcle provides certain instructions which enables us to efficiently implement switch-blocking:

1. **RDWIM** : This is a privileged instruction that reads the **WIM** register contents into a register.

2. **WRWIM** : WIM can be written by this instruction.

3. **NEXTF** and **PREVF** These instructions are similar to **SAVE2** and **RESTORE2** instructions, however the new **CWP** (Current Window Pointer) is determined by the next alternate window which is enabled i.e. for a **PREVF**, **CWP** - 2 or 4 or 6 is performed based on which window is enabled. First **CWP** - 2 is attempted and if that window is not enabled, **CWP** - 4 is attempted, etc. If no window is enabled, **CWP** remains the same. The add behaviour as in **SAVE** and **RESTORE** instructions is preserved and no traps are taken. Analogously **NEXTF** performs **CWP** + (2,4,6). Both **NEXTF** and **PREVF** are 1 cycle instructions. The windows are enabled only if the **WIM** bits of the corresponding context are set to 1.

ASIM was modified in order to simulate these instructions.

4.1 Context Switching Mechanism

The following pseudocode represents the context switching mechanism.

```
;; switch-blocking context switch
;; keep WIM mask bits of user frame in th22
;; e.g.: 00110000 for context 2
 (rdwim th16)
 (wrwim th22 th16)
   ; toggle invalid bits of current context
 (rdpsr th16)    ; save PSR
 (nextf zero zero zero)
   ;go to next executable trap frame
 (wrpsr th16 zero)
 (jmpl (d@r thtpc 0) zero)
 (rett thtnpc zero)
;; END of context switch.
```

4.2 Handling Synchronization Requests in Switch-blocking

Switch-blocking is a signaling mechanism. Hence the disabled thread has to be notified by another thread that the request which caused it to wait has been satisfied.

Alewife supports synchronization through full/empty bits and traps. The full/empty bit automatically serves as lock for its associated memory location. Hence all synchronizations are based on acquisition and release of locks. This method can be used for efficient handling of failed synchronization requests. The various synchronization constructs are handled as follows.

1. **Semaphores and Mutex**: These are almost identical in their implementation. Each of them has a value slot, a full/empty bit, and a queue slot. If the full/empty bit is 0, it indicates that a thread is accessing the semaphore or the mutex value and therefore the other threads which want to access this value will be queued up.

   Hence when a thread finds a value of 0 in the full/empty bit a context switch is effected and the thread is put on the queue associated with that location. When the full/empty bit is set to 1, the first thread on the queue is allowed access to the value, the context corresponding to this thread is restored and **WIM** bit of this context is reset to 0.

2. **J-structures and L-structures:** If the full/empty bit of a J-structure or a L-structure is 0, the reader is forced to wait. The writer will write the value into the J-structure and set the full/empty bit to 1 and releases any readers waiting on it.

   Hence for a J-structure or a L-structure, if the full/empty bit is 0 for the reader, **WIM** bit of the requesting thread is set to 1. The requesting thread will be queued up. Once the writer has set the full/empty bit to 1, all the threads on the queue will have their **WIM** set to 0.

3. **Barriers**: Let M be the total number of participants in the barrier. Then the first $M-1$ threads will have their **WIM** bits set to 1. The arrival of the M^{th} thread will reset the **WIM** bits of the remaining $M-1$ threads.

4. **Futures**: A thread that requires the value of a future is a consumer that is synchronizing with the thread producing that value. Consumers will have to wait for unresolved futures. When the value is available, the thread that produced the value will release all those threads that are waiting for that value.

   Hence when a future is in an unresolved state, that is the value of the expression, **(future X)**, is not available, the **WIM** bit is set to 1. Once the expression is evaluated, the **WIM** bit is reset to 0.

4.3 Enabling and Disabling of Contexts

4.3.1 Remote Memory Accesses

Initially, the **WIM** of an active context is 0. When a thread issues a remote-memory-access, a *cache miss trap* is generated. In the trap handler, the state of the context is saved and the **WIM** bits of that context are set to 1. Next, a context switch is effected by the use of the **NEXTF** instruction which searches for a context whose **WIM** bits are 0. When the remote-memory-access is serviced, the **WIM** bits corresponding to that context are set to 0 and the context is enabled.

To reset WIM in hardware so that no polling is necessary, additional pins are needed on the chip: namely, the hardware context to be enabled and an enable signal. Due to the pipelining in the chip, this reset takes effect only after a delay of 4 cycles. We assume that the Alewife controller has been modified so that when the remote memory access is complete, in addition to providing the data, the controller sets the enable pin along with the context number.

For computing the service time of a remote-memory-access, the following network delay model has been used [AGA91]:

$$T_c = (1 + \frac{\rho B(k_d - 1)}{(1-\rho)k_d^2}(1 + \frac{1}{n}))nk_d + B$$

Where

T_c	=	Network latency, for memory transactions.
ρ	=	Channel utilization.
B	=	Message size in flits.
n	=	Dimension of the network.
k_d	=	Average distance a message must travel in each dimension.

Since there is a delay of 4 cycles due to the pipeline in SPARCLE before the enabling of a hardware context, the effective memory delay as seen by the processor is 4 cycles longer. In this paper, the memory times given include this extra 4 cycles.

4.3.2 Handling Remote Signalling

For remote signalling, we have assumed that the Alewife controller has additional hardware to enable hardware contexts when a synchronizing condition is satisfied. Since all synchronization constructs are implemented using full/empty bits, the failed thread would wait (spin-waited in switch-spinning) on the local copy of the synchronization location in the remote access cache. When the full/empty bit is changed, a cache coherence transaction ensues; for an efficient switch-blocking implementation, we have assumed that this transaction has enough information to set a hardware bit that enables the context as in the remote memory access. This is definitely feasible as the current Alewife controller has a remote access cache where pending transactions have entries.

4.3.3 Overheads

Assuming the above hardware support, the additional overhead involved in a context switch for an implementation of switch-blocking are the instructions that set and reset the **WIM** bits. Two instructions **RDWIM** and **WRWIM** are executed in succession to toggle the **WIM** bits of the context. Hence

$$C_{sb} = C_{sp} + 2 = 16$$

.

5 Results and Analyses

In order to demonstrate the performance of the switch-blocking waiting algorithms, simulations were run on ASIM (Alewife's Simulator). The following waiting algorithms were considered for the simulation and Table **??** lists the default parameter:

1. Switch-spinning.
2. Switch-spinning/block.
3. Switch-blocking.
4. Switch-blocking/block.

5.1 ASIM

ASIM is a cycle-by-cycle simulator that simulates the processor, the memory system and the interconnection network. The organization of this simulator is as shown in Figure **??**. In order to execute a bench mark, the program is compiled and linked with the run-time system to produce executable object code. The APRIL simulator executes the machine instructions in the object code. The cache simulator is responsible for modeling the state of the cache and the cache coherence protocol. The network simulator is used to simulate the network latency incurred whenever a cache transaction involves communication with a remote cache.

ASIM can be run in two modes. Whenever an instruction that requires a memory transaction is executed, the APRIL simulator can choose to consult the cache simulator or not. In the *complete* mode, the cache simulator is consulted and proper cache behavior is modeled. In the *processor-only* mode the cache simulator is not consulted and a cache hit is assumed. This effectively simulates an ideal shared-memory system with fixed memory latencies. This mode is sometimes needed to filter out the performance benefits of memory latency masking.

5.2 Benchmark Programs

When the number of concurrently runnable threads is guaranteed not to exceed the number of hardware contexts available to execute them, the degree of threading is less than or equal to 1, and we say that it is *matched*. When the number of concurrently runnable threads is allowed to exceed the number of contexts, the degree of threading can be greater than 1, and we say that it is *unbounded* or *unmatched*. Both matched as well as unmatched benchmarks were run on Alewife.

The benchmark programs representing the three basic synchronization types were run on Alewife. Following is a description of the benchmarks ordered by synchronization type.

5.2.1 Mutual Exclusion

Mutex : distributes worker threads evenly throughout the machine. Each thread runs a loop that with some fixed probability acquires a mutex, does some computation, then releases the mutex.

5.2.2 Barrier Synchronization

CGrad : The conjugant gradient method [BER89] is a numerical algorithm for solving systems of linear equations. The data structure for the 2-D grid is mapped evenly among processing nodes in a block fashion: this mapping reduces the amount of communication between nodes.

The computation involved in each iteration of **CGrad** is a matrix multiply, two vector inner products, and three vector adds. The matrix multiply involves only nearest neighbor communication because Poisson's equation results in a banded matrix. Each inner product involves a global accumulate and broadcast. Because of the need to compute vector inner products which involve accumulate and broadcast, it is natural for **CGrad** to use barrier synchronization.

5.2.3 Producer-Consumer

MGrid applies the multigrid algorithm [BER89] to solve Poisson's equation on a 2-D grid with fixed boundary conditions. The 2-D grid is partitioned evenly among the processing nodes in a block-structured fashion. The multigrid algorithm consists of a series of Jacobi relaxations performed on grids of varying size.

Synchronization is implemented with J-structures. The 2-D grid is partitioned into subgrids, and a thread is assigned to each subgrid. The borders of each subgrid are implemented as J-structures so that threads responsible for neighboring subgrids can access the border values synchronously.

Cholesky factorization algorithms [ESW91] compute the **Cholesky** factor of a sparse, symmetric and positive definite matrix. The problem is expressed as $Ax = b$, where a vector x is to be determined, given a matrix A and a vector b. One method of solution is to determine a lower triangular matrix L, called A's *Cholesky* factor, such that $A = LL^T$, and then solve the two triangular linear systems $Ly = b$ and $L^T x = y$. Here also, the data structure is mapped in a block fashion.

Eshwar et. al. [ESW91] have proposed two algorithms:

1. CFan-in (Cholesky Fan-in algorithm): all the needed columns on the left are collected at the current column

2. CFan-out (Cholesky Fan-out algorithm): all the columns that need the current column are sent the data

5.3 Simulation Results

The simulation results that were obtained by running ASIM on the different waiting algorithms are tabulated in this section. The two-phase algorithms were run with $\alpha = 1$. The message size (B) was assumed to be fixed at 8 flits.

In all the simulations, the memory access time was varied from 8 to 40 processor clocks. In the case of matched benchmarks, this time was varied from 8 to 200. A high value of 200 was chosen for some simulations as in the case of the current DEC 10000 Alpha multiprocessors, the bus access time is already 340 ns (68 pclocks) and this does not take into account the delays due to networking. Hence, a memory access time of 200 is not an unreasonable projection for a remote-memory-access in some future designs.

We will compare switch-spinning against switch-blocking and the two-phase switch-spin/block against two-phase switch-block/block. All the benchmarks were written in the Mul-T language. Complete and exhaustive simulations have been done only for multi-grid.

Mutex The simulation results for **Mutex** are presented in Tables **??** and **??** for the matched case. These simulations were run with a queue length of 32 (i.e. $M = 32$).

MGrid The simulation results for **Mgrid** are presented in Tables **??** and **??** for the matched case, and in Tables **??** and **??** for the unmatched case. These simulations were run on a 64 x 64 grid.

CFan-in The simulation results for **CFan-in** are presented in Tables **??** and **??**. These simulations were run on a 128 x 128 matrix.

CFan-out The simulation results for **CFan-out** are presented in Tables **??** and **??**. These simulations were run on a 128 x 128 matrix.

CGrad The simulation results for **CGrad** are presented in Tables **??** and **??** for the matched case, and in Tables **??** and **??** for the unmatched case. These simulations were run on 64 x 64 grid.

5.4 Discussion

- The model presented in Section 3.1 is partly corroborated by the simulation results.

  Figures **??**, **??**, **??**, and **??** illustrate the plots of the probablity of i contexts being alive. This probability was computed by summing the time for which i contexts were enabled and dividing by the total execution time. It can be seen that the graph has a exponential tail similar to Fig **??**, hence corroborating the model developed in Section 3.1. It can also be noted that in the graph obtained for **CGrad** resembles normal distribution, instead of the exponential distribution in the other cases. In a barrier synchronization all the participating threads have to be blocked, hence the frequency with which all the contexts are disabled is more.

- There is a slight decrease in the advantage of switch-blocking over switch-spinning as memory access time is increased from 8 to 40 in many of the simulations before it increases again after 40. This is very likely due to the increased overhead of 2 extra cycles in switch blocking that shows its effect as long as the number of failed context switches is small. Once the latter becomes larger with larger memory access times, the extra overhead of 2 cycles is masked by the efficiency of switch-blocking.

- Table **??** lists the average number of cycles wasted by the processor when all the contexts are disabled. The last column lists the percentage of time wasted by the processor. It can be seen that this time is negligible when compared to the total execution time, hence the overhead involved in the implementation of switch-blocking is small compared to its performance.

- Table **??** lists the overhead involved in the case of switch-spinning, taking remote memory access into account. As the maximum percentage of wastage even for a channel utilization of 0.90 and a memory access time of 1000 is about 9.30%, remote memory accesses do not play a major role in the performance of switch-spinning. The benchmark that we have considered in Table **??** is the matched **MGrid**. The total number of memory accesses in the program was 64, 918.

6 Conclusions

Simulation results have shown that switch-blocking improves the performance of Alewife by about 6-8% over switch-spinning for the matched case and between 3-4% for the unmatched case. The hardware overhead involved in the implementation is nominal as instructions and registers already provided by Sparcle were used except for an additional line from the memory subsystem into the Sparcle chip for handling switch-blocking due to long-duration remote memory accesses. Some of the conclusions are listed below.

- An increase in memory access time improves the performance of switch-blocking only marginally, indicating that remote-memory accesses do not play a major role in the performance enhancement. For this reason, even though an increase in network channel utilization leads to an increase in the network latency, the performance of switch-blocking is improved only marginally.

- The two-phase switch-block/block algorithm performs better than the two-phase switch-spin/block algorithm by about 6-8% for the matched case.

- For the unmatched case, both switch-spin/block and switch-block/ block perform much better than switch-spinning. When the degree of threading is unbounded, a potential for deadlock exists if polling is used exclusively since we cannot guarantee that the thread waited upon is not unloaded. Hence the two-phase switch-block/block and switch-spin/block outperform their single-phase counterparts.

- The current Alewife architecture uses the sequential consistency model. It is clear that going to any other model (like weak consistency) can reduce the number of context switches and thus increase performance but the processor very rarely is completely idle from the above results. Hence, reducing the context switch time may be a better and simpler way of increasing performance than going in for a more complex memory consistency model due to the complexity of such an implementation and the small payoff.

References

[CHA91] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and David Yeaung. The MIT Alewife Machine : A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessor.* Kulwer Academic Publisher, 1991. An extended version of this paper appears as MIT/LCS Memo TM-454, 1991.

[SMI81] B. J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *SPIE, 298:241-248, 1981*

[KRA90] Anant Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz. APRIL : A Processor Architecture for Multiprocessing. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

[KUB91] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories : A Scalable Cache Coherence Scheme. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*(ASPLOS-IV). ACM, April 1991.

[CHA90] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache Coherence in Large-Scale Multiprocessors. *IEEE Computer*, June 1990.

[MOH90] Eric Mohr, David A. Kranz, and Robert H. Halstead. Lazy Task Creation: A Technique for Increasing The Granularity of Parallel Programs. In *Proceedings of Symposium on Lisp and Functional Programming*, June 1990.

[HAL90] Robert H. Halstead. Multilisp: A Language for Parallel Symbolic Computation. *ACM Transactions on Programming Languages and Systems,* 7(4):501-539, October 1985.

[LIM91] Beng-Hong Lim. Waiting Algorithms on Large Scale Multiprocessors. *MIT/Technical Report.* Feb. 1991.

[BER89] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation,* chapter 2. Prentice Hall, 1989.

[AGA91] Anant Agarwal. Limits on Interconnection Network Performance. In *IEEE Transactions on Parallel and Distributed Systems*, 1991.

[ESW91] Kalluri Eswar, P. Sadayappan and V. Visvanathan. Parallel and Direct Solutions to Sparse Linear Systems. In *Proceedings of Parallel Computing on Distributed Multiprocessors.* , 1991.

[KAR90] A. Karlin et.al., "Competitive Randomized Algorithms for Non-Uniform Problems," Procs. of Ist Annual ACM-SIAM Symp. on Discrete Algorithms, Jan 1990.

[Dash]

[Dash91]

[NET87] D. Bertsekas and R. Gallager, "Data Networks," Prentice-Hall, 1987