Journal of Universal Computer Science, vol. 6, no. 10 (2000), 928-947
submitted: 16/3/00, accepted: 10/10/00, appeared: 28/11/00

Performance of Switch Blocking on Multithreaded Architectures

K. Gopinath
Department of Computer Science & Automation
Indian Institute of Science, Bangalore
gopi@csa.iisc.ernet.in

Krishna Narasimhan M.K.
Department of Computer Science & Automation
Indian Institute of Science, Bangalore

Abstract: Block multithreaded architectures tolerate large memory and synchronization latencies by switching contexts on every remote-memory-access or on a failed synchronization request. We study the performance of a waiting mechanism called switch-blocking where waiting threads are disabled (but not unloaded) and signalled at the completion of the wait in comparison with switch-spinning where waiting threads poll and execute in a round-robin fashion. We present an implementation of switch-blocking on a cycle-by-cycle simulator for Alewife (a block multithreaded machine) for both remote memory accesses and synchronization operations and discuss results from the simulator. Our results indicate that while switch-blocking almost always has better performance than switch-spinning, its performance is similar to switch-spinning under heavy lock contention. Support for switch-blocking for remote memory accesses may be appropriate in the future due to their strong interactions with synchronization operations.

Key Words: algorithms, performance, theory. barriers, blocking, competitive analysis, locks, producer-consumer synchronization, spinning, waiting time
Category: C.1.2, C.4, D.4.1, D.4.8

1 Introduction

Remote memory access latencies are increasingly becoming high in large-scale multiprocessors [1, 13]. 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 [14]. A similar problem arises when the processor must wait for a synchronization event; the problem here 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 [7] machine implements this solution by using a processor that can switch rapidly between multiple threads[1] of computation and a cache controller that supports multiple outstanding requests. Processors that

[1] A *thread* is a process with its own processor state but without a separate address space. A *hardware context* is a set of registers on a processor that implements the processor-resident state of a thread. A *context switch* is a transfer of processor control

switch rapidly between multiple threads of computation are called *multithreaded architectures*. 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.

In the prototypical *finely* multithreaded machine HEP [8] or its more recent version TERA [2], the processor switches between processor resident threads every cycle. Architectures employing such cycle-by-cycle interleaving of threads are called for *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 may force a context switch depending on the waiting mechanisms chosen: spinning, switch-spinning, blocking and switch-blocking.

Current multiple issue superscalar processors exploit parallelism inside a single thread and not across threads, especially without the ability to tolerate latencies due to synchronization or cache faults etc. Simultaneous multithreading leverages the register renaming mechanism within dynamically-scheduled superscalar processors to allow instructions from multiple threads to be active simultaneously within the pipeline [4]. However, in block multithreading, instructions are executed from within a single thread until the next context switch, for higher single-thread performance..

1.1 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. It is possible to spin-wait economically on a machine with hardware cache coherence like Alewife by using cache invalidations to inform the waiters of a change in the synchronization variable.

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.

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.

from one processor-resident thread to another processor-resident thread. No thread state needs to be saved out into memory. A thread is *running* if it is resident in a hardware context regardless of whether it has control of the processor.

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.

The choice between waiting mechanisms depends on several factors like the expected wait time, the blocking overhead, whether the number of concurrently executable threads in a program does not exceed the number of hardware contexts (matched) or is unbounded (unmatched), etc. In addition, there are hybrid strategies like two-phase algorithms that employ one mechanism (say, spin) during the first phase and if the synchronization condition is not satisfied within the first phase, employ another mechanism (say, blocking). In the unmatched case (more threads than contexts), if there are more active threads than contexts, they get unloaded in pure blocking waiting algorithms. In the switch-block case, they get unloaded in a 2-phase algorithm after B time units (the cost of unloading and loading a hardware context). In 1-phase switch-block, they do not.

Previous work at MIT [10] has studied the performance of the first three waiting mechanisms under various models (2-phase algorithms, matched *vs* unmatched, etc.). Here we report the modelling, implementation and benchmark performance of *switch-blocking* and compare it with switch-spinning. Switch-blocking requires modest additional hardware resources on top of what is required for switch-spinning. Our results indicate that while switch-blocking almost always performs better than switch-spinning, its performance is similar to switch-spinning under heavy lock contention. Support for switch-blocking for remote memory accesses may be appropriate in the future due to their strong interactions with synchronization operations.

1.2 Background and Related Work

Beng-Hong Lim and Anant Agarwal [10] studied waiting algorithms for synchronization in large-scale multiprocessors. Their work addresses switch-spinning in 1-phase and 2-phase versions.

SUN has recently released a MAJC processor, MAJC-5200, that also uses block multithreading; this processor is designed for Java. This chip exploits parallelism through multiple CPUs (2) per chip, "vertical micro-threading" (same as block multithreading) through low overhead context switching triggered through long latency memory fetch, etc. and finally the use of VLIW in each thread [3].

One future architecture that plans to use block multithreading is "Blue Gene" from IBM, a massively parallel system currently being designed to have more than 8 million simultaneous threads of computation [17] to model the folding of human proteins. This architecture is being referred to as SMASH (Simple, Many and Self-Healing) as it will also be self-stabilizing and self-healing, ie. automatically be able to overcome failures of individual processors and computing threads. Blue Gene is expected to consist of more than one million 1 gigaflop processors with 32 of them on a single chip and 8 threads per processor.

1.3 Roadmap

In Section 2, we propose a Markov model for switch-blocking to estimate the performance advantage of switch-blocking over switch-spinning. Section 3 discusses two-phase waiting algorithms, as handling deadlocks in pure switch-blocking or switch-spinning can result in considerable loss of performance (due to use of coarse timeouts). In Section 4, we describe briefly Alewife's architecture and synchronization constructs followed by implementation details of switch-blocking. Section 5 presents the results of running various benchmarks on Alewife's simulator (ASIM) [5, 6] comparing one and two phase switch-blocking with switch-spinning and some discussion on the effectiveness of the models. ASIM (Alewife simulator) is a cycle-by-cycle simulator developed at MIT and we have modified it to run switch-blocking. It has a functional simulator for the memory subsystem, though. Finally, we conclude with some observations and future work.

2 Model for Switch-Blocking

Let T denote the random variable for wait time and $f(t)$ the probability density function (PDF) for wait time. Let B be the cost of blocking. If t is the wait time under spinning, define γ as the relative efficiency of switch-blocking over spinning so that t/γ is the waiting cost under switch-blocking for the same wait. The cost is modelled as being proportional to t as leaving contexts disabled diminishes the latency hiding efficiency (as experienced by other threads) compared to switch-spinning. In addition, a switch-blocked thread waiting for access to some lock will, upon being woken up one or more times, introduce network traffic that is proportional to the waiting time.

The parameter γ is not easy to estimate in closed form in contrast to β, the relative efficiency of switch-spinning, which is approximately given by [10]

$$\beta = \frac{t}{(x + C_{sp})\lceil \frac{t}{N(x+C_{sp})} \rceil} \tag{1}$$

Here N is the total number of hardware contexts, C_{sp} the context switch time for switch-spinning and $\lceil \frac{t}{N(x+C_{sp})} \rceil$ 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. However, γ is very close to the value of β as the simulations show a difference in performance that is less than 5%-8%.

To investigate the performance difference between switch-spinning and switch-blocking, we develop the following combined Markov model for both 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 probabilities of occupancy of the various states and the sojourn times computed in the combined Markov model to then estimate the additional overhead of switch-spinning over switch-blocking. (We leave the modelling for the unmatched case as future work.)

2.1 Combined Markov 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.

The Markov model (Figure 1) has $N+1$ states where i contexts are enabled in state i. All the contexts are disabled in state 0. Let π_i denote the steady state probability of state i. Let $E[T_i]$ denote the expected time when i contexts are enabled.

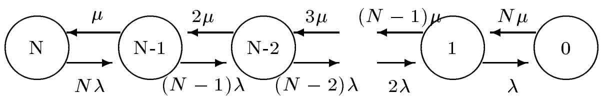


Fig 1: *Markov model for switch-blocking.*

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 one context is enabled from the remaining i contexts. 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 rate at which one context is disabled from the remaining $(N-i)$ contexts. Hence this rate is $(N-i)\lambda$. The steady state probabilities can be calculated as follows:

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

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

$$\ldots$$

Hence, $\pi_{(N-i)} = \frac{N!}{i!(N-i)!}(\frac{\lambda}{\mu})^i\pi_N$ 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 2:

$$E[T_{N-i}] = \frac{1}{(N-i)\lambda + i\mu} \tag{2}$$

Substituting $i = N$ in Equation 2, the expected time all contexts are blocked is given by

$$E[T_0] = \frac{1}{N\mu} \tag{3}$$

The PMF's for different values of λ/μ are plotted in Figures 2 and 3 whereas PMF's through simulations for multigrid and conjugate gradient (see Section 5 for information on these benchmarks) are plotted in Figures 4 and 5.

2.2 Modelling the Difference

To model the difference between switch-spinning and switch-blocking, we make the assumption that the probabilities of occupancy and sojourn times of the various states in the Markov model are the same for both and given by the above equations with the only difference being the extra context switches suffered by the former. This approximation is reasonable as the difference in the simulated times for many benchmarks is typically not more than 8% and the current hardware implementation can scan for an available context in the same sequence in both the waiting mechanisms.

The extra overhead can be modelled as the extra waiting time in a M/G/1 queue with vacations [15] with the vacations being the extra context switches suffered by switch-spinning: here W is the wait time, X is the service time and V the vacation:

$$W = \lambda\overline{X^2}/2(1-\rho) + \overline{V^2}/2\overline{V} \tag{4}$$

Let the service time include the context switch time. The vacation time in switch-spinning is given by kC_{sp} where k is the number of idle contexts attempted before success. Switch-blocking has, with each context-switch, a fixed short vacation given by $C_{sb} - C_{sp}$, where C_{sb} is the context switch time for switch-blocking.

The vacation part of the waiting time in Equation 4 is computed as follows $(N = 4)$. Given that the number of enabled contexts is i, and the current context is enabled, the average (variance) is computed by enumerating all the possible states of the contexts, and, assuming that they are equiprobable, multiplying by the time (square of the time) for the number of intervening failed contexts.

$$\begin{aligned}\overline{V_{sp}} &= C_{sp}(1-\pi_0)(\pi_4 * 0 + \pi_3/3 + \\ &\quad \pi_2(1/3+2/3) + 3\pi_1) + \pi_0 E[T_0] \\ &= xC_{sp}(1-\pi_0) + \pi_0 E[T_0]\end{aligned} \tag{5}$$

$$\begin{aligned}\overline{V_{sp}^2} &= C_{sp}^2(1-\pi_0)(\pi_4 * 0 + \pi_3/9 + \\ &\quad \pi_2 + 9\pi_1) + \pi_0 E[T_0]^2 \\ &= yC_{sp}^2(1-\pi_0) + \pi_0 E[T_0]^2\end{aligned} \tag{6}$$

$$\overline{V_{sb}} = (1-\pi_0)(C_{sb} - C_{sp}) + \pi_0 E[T_0] \tag{7}$$

$$\overline{V_{sb}^2} = (1-\pi_0)(C_{sb} - C_{sp})^2 + \pi_0 E[T_0]^2 \tag{8}$$

$$W_{extra} = \overline{V_{sp}^2}/2\overline{V_{sp}} - \overline{V_{sb}^2}/2\overline{V_{sb}} \tag{9}$$

The extra context switches suffered by switch-spinning is given by W_{extra} and this enables us to compute γ from Eq. 1. In addition, the specific contribution to the overhead for switch-spinning from memory transactions can be computed in a similar fashion.

2.3 Evaluating the Model

To estimate the effectiveness of the Markov model, consider the case for $\lambda/\mu = 0.5$ that is close to the observed probability 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 equation 3 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 Equation 4 as $0.69C_{sp}$ 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: 240,000 cycles; observed: 330,000 cycles). This is quite reasonable as we have not explicitly taken into account the difference in the context switches due to synchronization or remote cache misses and we have made many simplifying assumptions.

If probabilities computed from simulations are used instead of using calculated values from the Markov model (but still using Eq.9), we get a more accurate answer. Using the values in Figure 4, the extra overhead is 0.97 context switches (for a total of 337,400 cycles), which is closer to the observed value.

3 Two Phase Algorithms

As deadlocks are possible in the context of unmatched algorithms, pure switch-spinning and switch-blocking may not be suitable as the overhead of deadlock detection (typically by timeouts) can be substantial. Two phase algorithms can handle this problem with ease in addition to the original motivation of tackling widely varying waiting times of synchronization events.

Without any information about the distribution of wait times of synchronization events and remote memory references, one cannot expect an on-line waiting algorithm to perform as well as an 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 et al. [12] 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 length 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. Typically, 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 the length of the switch-blocking phase be 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 various types of waiting algorithms considered in our simulations are as follows:

Always Switch-Block : This is a pure 1-phase strategy with the cost of waiting for t cycles being t/γ.

Blocking : The cost of blocking is B, regardless of the wait time distribution.

Optimal Switch-block/Block : The cost of switch-blocking in phase 1 is reduced by a factor of γ with the maximum length of the phase being γB.

Two-phase Switch-Block/Block : This algorithm switch-blocks until the cost of switch-blocking is equal to αB and then blocks if necessary.

Once the wait time distributions for commonly used synchronization types (like mutual exclusion, barriers, producer-consumer) are derived by making suitable assumptions (see Lim [10] for details), one can compute the cost of the different waiting algorithms. Assuming that the factor for switch-blocking is γ instead of β, all the results that Lim derives are carried through except that β is replaced by γ.

4 Implementation of Switch Blocking in Alewife

Alewife is a cache-coherent, block-multithreaded, distributed memory multiprocessor that supports a shared-memory programming abstraction. The initial implementation of the SPARC-based node processor is SPARCLE (also called APRIL) [9] and designed to meet 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.

Register windows in the SPARC processor permit a simple implementation of block multithreading. Two register windows are allocated to each thread. The Current Window Pointer (CWP) 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 CWP to a new register window for a total of 14 cycles. Through the use of *memory exception* (MEXC) line on SPARC, the controller can invoke synchronous traps and rapid context switching.

SPARC provides a 32 bit register, the *window invalid mask* (WIM), whose bits indicate whether a context is enabled (0) or not. The SPARCLE processor is based on the following modifications to SPARC architecture: emulation of multiple FP hardware contexts, detection of unresolved futures through SPARC word-alignment and tagged-arithmetic traps. In addition, it has the following instructions[2] that enables us to efficiently implement switch-blocking: RDWIM is

[2] SPARCLE also has SAVE2 and RESTORE2 that change CWP by two.

a privileged instruction that reads the WIM register contents into a register while WRWIM writes WIM. In NEXTF and PREVF, the new CWP is determined by the next alternate window that is enabled. In PREVF, CWP - 2, 4 or 6 is checked in sequence to locate an enabled window. 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 checks CWP + 2, 4 or 6. Both NEXTF and PREVF are 1 cycle instructions. ASIM (Alewife's simulator: see Section 5) has been modified to simulate these instructions and registers like WIM. According to the SPARC V8 standard [16], WRWIM can have 0-3 delay slots depending on the implementation with NEXTF to be executed only after these delay slots for a correct operation[3]. Our implementation has assumed a delay slot of 1.

4.1 Enabling and Disabling of Contexts

First, we discuss one important boundary case. If all the contexts are disabled in the matched case (i.e., when all contexts contain switch-blocked threads), the processor idles[4].

4.1.1 Handling Remote Memory Accesses

Initially, the WIM bits of an active context are 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 [11]:

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

$$\begin{aligned}
T_c &= \text{Network latency for memory transactions} \\
\rho &= \text{Channel utilization} \\
S &= \text{Message size in flits (8 bits)} \\
n &= \text{Dimension of the network (2)} \\
M &= \text{Memory access time} \\
k_d &= \text{Average distance a message must travel} \\
&\quad\ \text{in each dimension}
\end{aligned}$$

[3] SPARCLE has a 3 cycle delay slot

[4] This is also the behaviour of the NEXTF and PREVF instructions in SPARCLE.

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.

4.1.2 Handling Synchronization

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 setting and resetting of full/empty bits. However, there is a fundamental difference between a cache miss trap and a full/empty trap. In the former, a memory transaction is pending when control returns to the processor after the cache miss trap. In the latter, the memory transaction has already completed before the full/empty trap is signalled. The above observations can be used to design a signalling system that does not need any additional hardware support; everything can be handled in software.

When a thread gets a full/empty trap, it queues its processor and context number in the queue slot associated with the empty location (in the same manner as when a thread gets blocked) before disabling the context it resides in. When another thread sets the location to full, it uses Alewife's messaging facility to send enable messages to the processors with switch-blocked threads on the wait queue. The message handler can then set the WIM bits appropriately.

For two-phase switch-block/block, when a thread decides to switch-block on a full/empty trap, it needs to queue both its thread-id and its context number in the queue slot. Keeping the thread loaded for exactly the blocking time is possible given that the current Alewife controller uses timers for its various needs. We need to dedicate 4 timers; control already exists. Before a context is switched out, the timer has to be initialized; this can be attempted in the 3 cycle delay in the current Sparcle after WRWIM for NEXTF to be executed. When the thread is woken up, the message handler also needs to check to see if the waiting thread has already blocked and relinquished the context. If not, it can set the WIM bits. Otherwise, it reenables the thread as when waking up a blocked thread. In our simulations, we have not incorporated the hardware timers (and the overhead from the associated traps) but let the simulator keep track of the time. The implementation specifics of each of the synchronization constructs is as follows:

Semaphores and Mutex: These are 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 bits of this context are reset to 0.

J/L-structures[5]: If the full/empty bit of a J/L-structure is 0, the reader is forced to wait. The writer will write the value into the J/L-structure and set

[5] 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

the full/empty bit to 1 and releases any readers waiting on it. Hence for a J/L-structure, if the full/empty bit is 0 for the reader, WIM bits of the requesting thread are 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 reset to 0.

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.

Futures: A thread that requires the value of a future **(future X)** is a consumer that is synchronizing with the thread producing that value and will have to wait if unresolved. 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, the WIM bits are set to 1. Once the expression is evaluated, the WIM bits are reset to 0.

4.1.3 Overheads

Assuming the above hardware support for handling remote memory accesses, the additional overhead involved in a context switch for an implementation of switch-blocking are the instructions that set and reset the WIM bits. As we have assumed a delay slot of one for WRWIM in our simulations, and RDWIM and WRWIM are executed in succession to toggle the WIM bits of the context, $C_{sb} = C_{sp} + 2 = 16$.

In the current SPARCLE implementation, with WRWIM's delay slot of 3, $C_{sb} = C_{sp} + 4 = 18$; this makes switch-blocking only marginally more expensive than our results indicate as switch-blocking attempts to reduce the number of context-switches. One additional overhead we have not modelled is due to traps from hardware timing counters when two-phase algorithms are used. As this is quite small compared to the blocking cost, our results are likely to be quite accurate. (As the initial loading of the timing counter can be done in the spare delay slot of the WRWIM of Sparcle, a more complete simulation for two-phase algorithms would need only to model the 3 cycle delay slot and the overhead from timing traps.)

5 Results and Analyses

To study the performance of waiting algorithms for switch-blocking *vs* switch-spinning, simulations were run on ASIM (Alewife's Simulator) for single and two phase versions. Table 1 lists the default parameters. The following waiting algorithms were considered for the simulation:

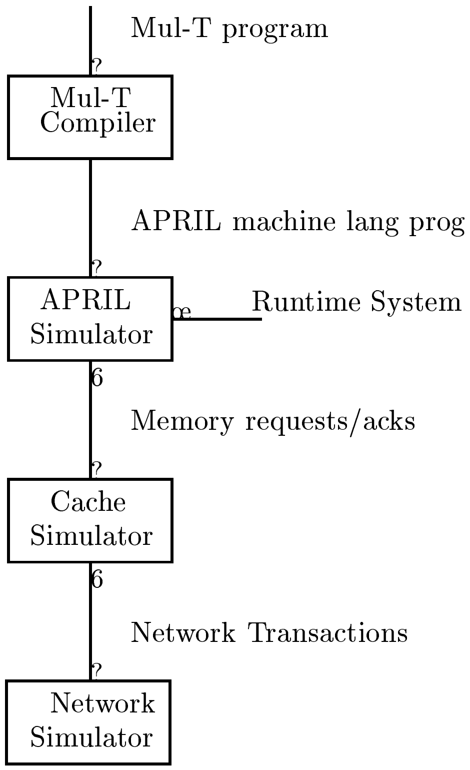
to 1 and releases any readers waiting on it. A J-structure can be *reset*. This clears out all the value slots and sets the full/empty bits to 0. Resetting a J-structure to an empty state enables efficient memory allocation and good cache performance.

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.

1. Switch-spinning (abbreviated as sspin in Figs 6-13)
2. Switch-spinning/block (abbreviated as sspin/bl)
3. Switch-blocking (abbreviated as sblock)
4. Switch-blocking/block (abbreviated as sblock/bl)

5.1 ASIM

ASIM simulates the processor, memory system and interconnection network cycle-by-cycle. The organization of this simulator is as follows:



In order to execute a benchmark, the program is compiled and linked with the run-time system to produce executable object code that ASIM executes. 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.

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

Parameter	Default setting
Number of Processors	64
Cache Coherence Protocol	LimitLESS (4 HW pointers)
Cache Size	64KB (4096 lines)
Cache Block Size	16 bytes
Number of Contexts	4
Network Topology	2-D Mesh

Table 1: Simulation Parameters.

threading is less than or equal to 1 or *matched*. Otherwise, the degree of threading can be greater than 1 or *unbounded* or *unmatched*. Both matched as well as unmatched benchmarks were run on ASIM. The following is a description of the benchmarks (all written in the Mul-T language) ordered by the three basic synchronization types:

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 and then releases the mutex.

Barrier Synchronization

CGrad uses the conjugate gradient method for solving systems of linear equations. The data structure for the 2-D grid is mapped evenly among processing nodes in a block fashion to reduce 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.

Producer-Consumer

MGrid applies the multigrid algorithm 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 of a sparse, symmetric and positive definite matrix by two methods: CFan-in (Cholesky Fan-in: all the needed columns on the left are collected at the current column) and CFan-out (Cholesky Fan-out: all the columns that need the current column are sent the data). For both, the data is mapped in a block fashion.

5.3 Simulation Results

The following simulation results were obtained by running ASIM to compare switch-spinning with switch-blocking and the two-phase switch-spin/block with two-phase switch-block/block. The two-phase algorithms were run with $\alpha = 1$ and $B = 1000$ cycles, the value of B being approximately the cost of saving and restoring a hardware context. Due to lack of space, we will be presenting graphs only for MGrid and CGrad, though the discussion will include the other benchmarks also.

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 DEC 10000 Alpha multiprocessors, the bus access time is 68 (processor) clocks and this does not take into account the delays due to networking.

MGrid: The simulation results for 64 x 64 grid are presented in Figures 6, 7 for the matched case, and in Figures 8, 9 for the unmatched case.

CGrad: The simulation results for 64 x 64 grid are presented in Figures 10, 11 for the matched case, and in Figures 12, 13 for the unmatched case.

5.4 Discussion

The model presented in Section 2 is partly corroborated by the simulation results. Figures 4 and 5 illustrate the plots of the probability of i contexts being alive. This probability is computed by summing the time for which i contexts are enabled and dividing by the total execution time. The MGrid graph has a shape that is similar to the shape predicted by the Markov model (Figure 3). However, the graph for CGrad resembles normal distribution, instead of the exponential distribution. In a barrier synchronization, all the participating threads get blocked at the barrier, hence the probability with which the contexts are disabled is higher.

It is interesting to note that there is improvement in the running time for switch-block/block over switch-block for matched mutex, MGrid, CFan-in and CFan-out but not for matched CGrad. The explanation is similar to the one that has been advanced by Lim [10] in the case of switch-spinning. In the matched case, a first expectation is for switch-block time to be lower than for switch-block/block as the overhead of loading and unloading is not present and the switch-blocked threads do not contribute to loading the network as they do not poll. However, this is not true in the context of heavy lock contention where the behaviour of the switch-block versions approaches that of the switch-spinning versions. In switch-blocking, whenever lock contention becomes sufficiently high, the switch-blocked threads that are enabled after a lock is released, possibly residing on different processors, try to acquire the lock and load the network heavily. However, switch-block/block would have a smaller loading of the network as the unloading & loading of the blocked threads (and their variance in time depending on cache hits) eliminates the possibility of bursts of memory reads to the same memory location[6]. Switch-block/block also does not incur the

[6] A similar situation arises for mutex [10] under blocking which works better than switch-spinning as the blocked waiters take longer to be reactivated and thereby avoid the detrimental effect of bursty lock requests.

overhead of blocking under low lock contention. The difference in performance is related to the extent of synchronization activity: higher in the case of mutex compared to MGrid, for example. In the case of matched CGrad, switch-block has a better performance over switch-block/block as expected. Similarly, in all the unmatched cases, switch-block/block performs better than switch-block as expected as deadlocks are handled better with blocking in the former rather than with expensive timeouts in the latter.

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.

Figure 14 lists the average number of cycles idled and the percentage of time idled by the processor when all the contexts are disabled. 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.

Figure 15 lists the number of times threads that issue remote memory accesses fail on the next try for matched MGrid employing switch-spinning (the percentage listed is that of this number with respect to context switches due only to remote memory accesses). Even when memory access is set to 8 cycles, there are quite a few failures as actual access may be in the region of 30-50 cycles with network overhead. The number of failures is related to the probability that the following three contexts are also not active; this is given by $\pi_1 = 0.18$ (from simulations) which is not negligible. Only latencies lower than $3C_{sp} = 42$ cycles can avoid failure with absolute certainty. It is also interesting to note from Figures 6, 7 that there is a sudden increase in the execution time at a memory access of 100 cycles. This is most likely due to the effective memory access (including network delays) being just about three times the context run length (43 from simulations for MGrid) and hence increasing the likelihood that the retry will not be successful. This is also connected with the sudden increase in the execution times at memory access times of 100 and above as the synchronization operations on remote locks involve memory accesses and they may have to be carried out multiple times (as local copies get invalidated by cache coherence transactions) before the operation is successful. This behaviour is seen on all the benchmarks.

As the difference of execution times between for access times of 8 and 200 in the case of multigrid from is between 4-5%, and as the difference increases sharply even more with higher access times (say, 400+) due to its impact on synchronization operations, switch-blocking for remote memory accesses may be judicious if the hardware resources are available. However, remote memory accesses do not play a major role in the comparative performance advantage of switch-blocking over switch-spinning at current range of memory access times. Similarly, with an increase in network channel utilization (leading to an increase in the network latency), the performance advantage of switch-blocking improves only marginally.

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 for the implementation is not substantial as instructions and registers already provided by SPARCLE have been used except for some additional lines (for an enable and for specifying the context number) from the memory subsystem into the SPARCLE chip for handling switch-blocking due to long-duration remote memory accesses and some support for hardware counters for the two-phase algorithms. Some of the conclusions are listed below:

- The two-phase switch-block/block algorithm performs better than the one-phase switch-block algorithm even for the matched case whenever there is heavy lock contention. Under low lock contention, the reverse is true.
- In the unmatched case, the two-phase switch-block/block algorithm performs better than the one-phase switch-block algorithm as deadlocks are handled better.
- 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 for one-phase algorithms 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.
- As memory access times are increased, the performance advantage of switch-blocking improves only marginally with respect to switch-spinning. Hence support for switch-blocking for remote memory accesses may not be judicious at current range of memory access times but may be so in the future due to its strong interactions with synchronization operations.

The current Alewife architecture uses the sequential consistency model. Other memory models (like weak consistency) can reduce the number of context switches and thus increase performance but the processor is very rarely 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. However, researchers at Stanford have found that more complex models give a reasonable payoff [14]. More work is needed to settle this question comprehensively.

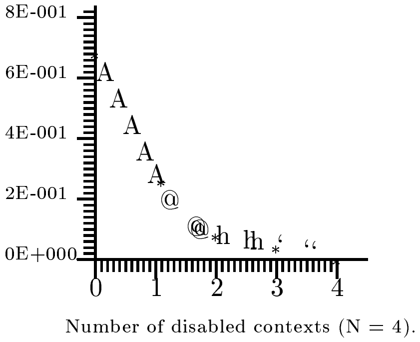


Fig 2: *PMF for* $\frac{\lambda}{\mu} = 0.1$

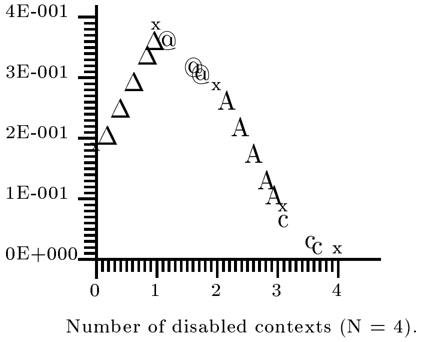


Fig 3: *PMF for* $\frac{\lambda}{\mu} = 0.5$

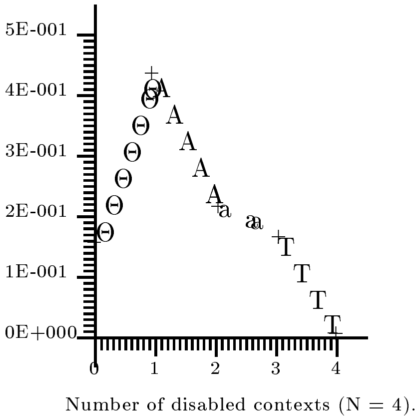


Fig 4: *PMF for MGrid* (64×64) *by Simulation*

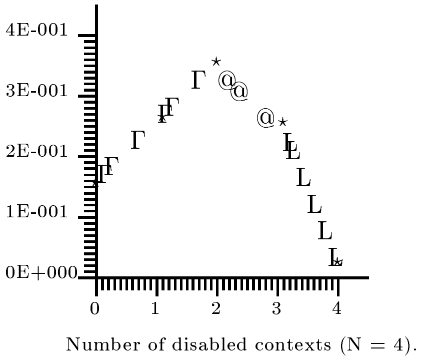


Fig 5: *PMF for CGrad* (64×64) *by Simulation*

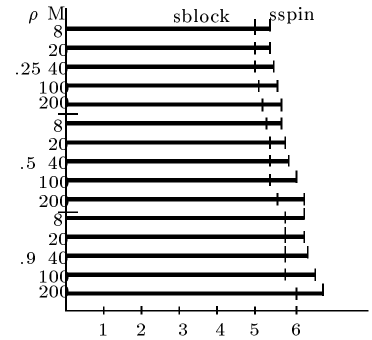


Fig 6: *MGRID m. 1-ϕ*

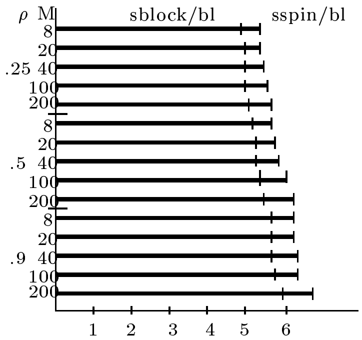


Fig 7: *MGRID m. 2-ϕ*

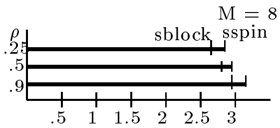


Fig 8: *MGRID unm. 1-ϕ*

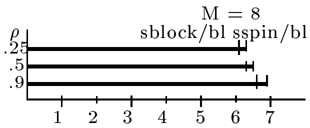


Fig 9: *MGRID unm. 2-ϕ*

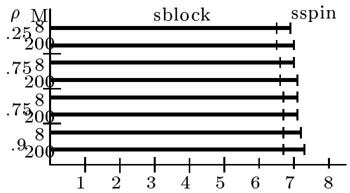


Fig 10: *CGRAD m. 1-ϕ*

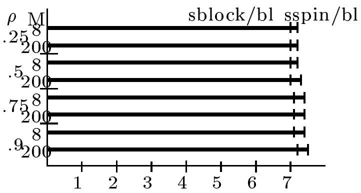


Fig 11: *CGRAD m. 2-ϕ*

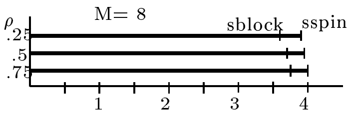


Fig 12: *CGRAD unm. 1-ϕ*

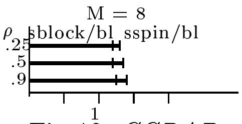


Fig 13: *CGRAD unm. 2-ϕ*

Figs 6-13: *Comparison of switch-blocking (sblock) vs switch-spinning (sspin) algorithms: 1-phase (1-ϕ) with switch phase only vs 2-phase (2-ϕ) with both switch and block (bl) phases; also matched (m.) vs unmatched (unm.). X-axis in Mcycles. M refers to memory access time.*

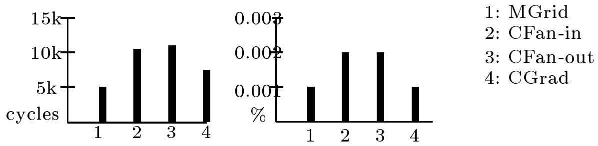


Fig 14: *Idling cycles with all contexts disabled and as % time*

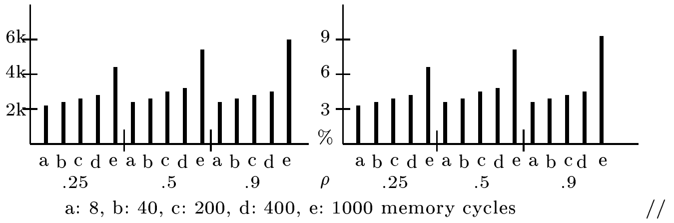


Fig 15: *Number of failed context switches for MGRID (see text)*

References

[1] Chris Holt, Mark Heinrich, Jaswinder Pal Singh, Edward Rothberg, and John Hennessy. The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors Technical Report CSL-TR-95-660, Computer Systems Laboratory, Stanford University, January 1995.

[2] Larry Carter, John Feo, Allan Snavely, "Performance and Programming Experience on the Tera MTA", *Proceeding SIAM Conference on Parallel Processing*, San Antonio, Texas (March 1999).

[3] S.Sudarshanan, "MAJC-5200: A High Performance Microprocessor for Multimedia Computing," LNCS 1800, p.163-170.

[4] D.M. Tullsen, S.J. Eggers, and H.M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," *Proceedings 22nd Annual International Symposium on Computer Architecture*, 1995

[5] David Kranz, David Chaiken, Anant Agarwal, "Multiprocessor Address Tracing and Performance Analysis," MIT VLSI Memo No. 91-624, Sep 1990. Available at http://www.cag.lcs.mit.edu/alewife/papers

[6] Beng-Hong Lim, "Instructions for Obtaining and Installing ASIM," Alewife Systems Memo #30, Dec'91. See also the website http://www.cag.lcs.mit.edu/alewife for later developments.

[7] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Ken Mackenzie, and Donald Yeung, "The MIT Alewife Machine: Architecture and Performance," *Proceedings 22nd Annual International Symposium on Computer Architecture*, 1995

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

[9] Anant Agarwal, B. H. Lim, D. Kranz and J. Kubiatowicz, "APRIL: A Processor Architecture for Multiprocessing," *Proceedings 17th Annual International Symposium on Computer Architecture*, p. 104-114, June 1990.

[10] B. H. Lim, Anant Agarwal, "Waiting Algorithms on Large Scale Multiprocessors," ACM Transactions on Computer Systems, 11(3):253–294, Aug. 1993.

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

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

[13] D. Lenoski, et al., "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," *Proceedings 17th Annual International Symposium on Computer Architecture*, p. 148-159, June 1990.
[14] A. Gupta et. al., "Comparative Evaluation of Latency Reducing and Tolerating Techniques," *Proceedings 18th Annual International Symposium on Computer Architecture*, p. 254-263, May 1991.
[15] D. Bertsekas and R. Gallager, "Data Networks," Prentice-Hall, 1987
[16] SPARC International, "The SPARC Architecture Manual Version 8,"Prentice-Hall, 1992
[17] IBM Research, http://www.research.ibm.com/bluegene/