Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessors

Bob Boothe* Abhiram Ranade*

Computer Science Division
University of California, Berkeley, CA 94720

Abstract

Shared memory multiprocessors are considered among the easiest parallel computers to program. However building shared memory machines with thousands of processors has proved difficult because of the inevitably long memory latencies. Much previous research has focused on cache coherency techniques, but it remains unclear if caches can obtain sufficiently high hit rates. In this paper we present improved multithreading techniques that can easily tolerate latencies of hundreds of cycles, and yet only require a small number of threads per processor. High performance is achieved by introducing an explicit context switch instruction that can be used by a simple optimizing compiler to group together several shared accesses. This grouping of shared accesses dramatically reduces the frequency of context switches compared to simpler multithreading models. The combination of our techniques achieves efficiencies of 80% or higher on a broad set of applications.

1 Introduction

Large shared memory multiprocessors will have very long latencies for remote memory accesses. For a 1024 processor machine, we expect latencies of hundreds of cycles[7, 9]. These long latencies are inevitable because of the size and complexity of any interconnection network that can connect thousands of processors and memories and support high bandwidth. Without some means of tolerating memory latency, a 1024 processor shared memory multiprocessor would spend a large fraction of its time waiting for shared accesses to complete.

Memory latency has traditionally been hidden by caches, and many researchers have tried to extend cache coherency schemes to allow cacheing to be used on scalable shared memory machines. [5, 9, 18]. Caches, however, do not actually hide memory latency, but instead, they hope to eliminate enough of the memory accesses that the latencies of the few remaining accesses will have only a small impact on the execution time. Consider a simple example where a program accesses shared memory every tenth cycle and where the memory latency is 200 cycles. Without any means of hiding latency, the program will execute for ten cycles and then wait 200 cycles for the result to return. The processor utilization will be only 5%. If we add a cache that can achieve a 90% hit rate, only every tenth access will have to wait, but that access will still wait 200 cycles. The efficiency has risen only to 33%. Recent simulation results from the DASH project[9] confirm this simple argument. They measured hit rates on shared-read references ranging from 66% to 80%, and processor utilizations ranging from 20% to 28% under a release consistency model. Other simulation studies have also reported low processor utilizations[5]. O'Krafka[18] reported more promising hit rates, but overall, cacheing alone does not appear to be an adequate solution to the latency problem.

Prefetching of data is another potential solution. If data can be prefetched far enough in advance, it will have returned from the memory network by the time the processor needs to use it. For many scientific codes prefetching works well[8, 9, 13]. But on less regular codes, such as those that traverse complex linked data structures, the sequence of references can not be predicted and prefetching will not be possible. Prefetching alone is not a general solution to the latency problem.

Multithreading is an alternative technique for hiding memory latency. If several threads are assigned to each processor, memory latency can be hidden by rapidly context switching to a different thread rather than waiting for a memory access to complete. While previous studies have shown mixed results when using multithreading as an adjunct to cacheing[1, 9, 25], in this paper we study multithreading on its own merits and find it to be very effective.

1.1 Overview

First we look at a simple multithreading model that context switches on every load from shared memory (Section 4). For a few applications the shared loads are spaced far enough apart that a small number of threads are able to hide the memory latency and utilize the available processing power. However for most applications, shared loads occur too frequently (as often as once every 5 cycles) and too close together (often in bunches) to allow a small number of threads

*This work is supported by the Air Force Office of Scientific Research (AFOSR/JSEP) under contract F49620-90-C-0029 and NSF Grant Number 1-442427-21936. Computing resources were provided by NSF Infrastructure Grant number CDA-8722788.

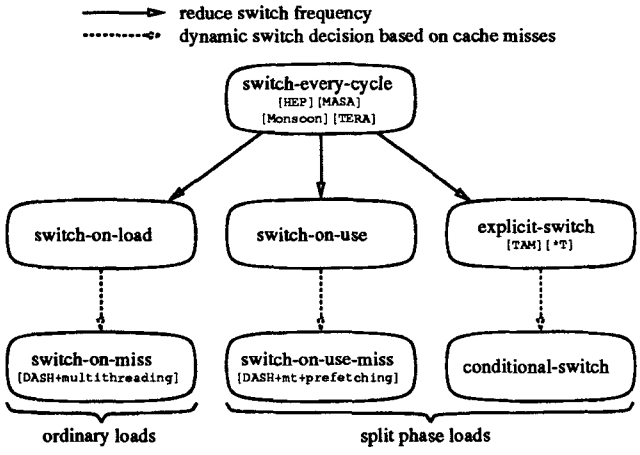


Figure 1: Evolution of Multithreading Models

to hide the latency. For these applications we need a better multithreading model.

For this next model (Section 5), we introduce the idea of grouping accesses together to eliminate unnecessary context switches. This grouping lets the processors wait for several accesses together rather than waiting for each access individually. The frequency of context switches decreases by as much as a factor of 5, and most of the closely spaced accesses can be grouped together. This one change allows all of the applications to execute with 70% or better efficiency on a multithreading machine with less than ten threads per processor.

The main problem with the previous model is that it requires a large bandwidth to shared memory. To address this problem we introduce a third multithreading model that adds a cache but still maintains the benefits of grouping (Section 6). For many of the applications this proves effective in both reducing the memory bandwidth and in further decreasing the multithreading level needed to achieve good performance.

Before we present our simulation results, we first present a structured overview of previous multithreading research (Section 2), and a description of the multiprocessor model and applications used in this study (Section 3).

2 Previous Work

Previous multithreading research has been motivated by four concerns: tolerating memory latency, building a fast pipeline, supporting a threaded execution model, and as an adjunct to cacheing. Figure 1 shows the evolution of multithreading models and some of the motivations for moving from one model to another. Some of these models have not been studied previously but can be predicted based on the motivations.

The costs of multithreading are the larger number of threads needed, the larger register file, the increased scheduling complexity, and the cycles lost to context switching overhead. These costs are influenced by when and how often context switching is performed.

The oldest model **switch-every-cycle** was used in the Denelcor HEP [14] and in MASA [11]. After each instruction, the processor switches to a different thread. This allows a fast CPU pipeline to be built because it eliminates data dependencies between instructions in the pipeline by interleaving different threads. It also allows memory latencies to be tolerated by not scheduling a thread until its references have completed. Unfortunately this model requires a complex scheduling mechanism and a large number of threads. By interleaving the instructions from many threads, a single thread is limited to a small fraction of the processing power. The Monsoon[20], P-RISC[17], and TERA[2] projects also switch every cycle, but have changed the scheduling policy so that scheduling decisions are made at a coarser granularity than every instruction.

With a normal RISC CPU pipeline, there is no need to context switch every cycle. The compiler can schedule instructions so as to hide the small pipeline delays. Context switches are needed only to hide the long memory latency of remote accesses.

The **switch-on-load** model switches on load instructions which access shared memory. Loads from local memory and other instructions all complete quickly, and can be scheduled by the compiler. Shared stores don't wait for their completion and therefore don't cause context switches either. The advantage of this model over **switch-every-cycle** is that a single thread can execute at full speed until it context switches. Fewer threads are needed to cover the latency since threads aren't being used to schedule individual instructions in the pipeline. Simpler hardware is needed since context switches are less frequent.

The **switch-on-load** model sometimes context switches sooner than it needs to. If a compiler can order instructions so that a load is issued several cycles before the value is used, the context switch does not have to occur until the actual use of the value. This allows a thread to hide some of its own memory latency by prefetching data. This mechanism is facilitated by using split phase instructions. The first instruction, `load`, issues the reference into the network, and a second instruction, `use`, waits for the result. The **switch-on-use** multithreading model context switches on the `use` instructions rather than the `load` instructions.

A benefit of the **switch-on-use** model is that several load instructions can be grouped together so that all of the loads in the group can be issued into the memory network before any of the results are used. This allows a single context switch (on the first of the uses) to wait for all of the loads in the group. For example, a simple computation may load two values from shared memory and then compute their average. Both loads should be issued into the memory network and then a single context switch performed upon the use of the first value. This allows waiting for both loads together rather than individually. The cost of this model is the addition of a `use` instruction for each shared memory load and hardware to count the results as they are returned.

An alternative method for grouping shared loads together is to add an explicit context switch instruction between the group of loads and their subsequent uses. This **explicit-switch** model is studied in this paper and is shown to eliminate from 50% to 80% of the context switches needed by the **switch-on-load** model. With fewer context switches, a smaller number of threads can be used to cover the latency. The most recent data flow research[6, 16] has adopted the **explicit-switch** model. Short threads execute until their completion at which point they cause a context switch to a new thread.

By adding caches to the previous models, the cache can satisfy many of the shared loads without going to shared memory. Only the shared loads that miss in the cache will have long latencies and cause context switches.

The **switch-on-miss** model switches at points where load instructions miss in the cache. An early study of this by Weber & Gupta[25] suggested substantial performance benefits were available, but a later study as part of the DASH project [9] had less optimistic results. Both of these studies restricted multithreading to at most 4 threads per processor. **Switch-on-miss** multithreading was also studied as part of the ALEWIFE project [1] and achieved good results with a few simple applications. One draw back of context-switching on cache misses is that the context switch is detected after a number of subsequent instructions have started down the CPU pipeline. These instructions must be canceled, and thus there will be a context switch cost of several cycles because of the wasted pipeline slots.

The **switch-on-use-miss** model context switches when a `use` instruction tries to use the value from a shared load that missed in the cache. The split phase instructions essentially provide a mechanism for prefetching data. The switch-on-use-miss model was studied (approximately) by the DASH project[9] when they looked at the combination of prefetching and multithreading. They found little benefit from prefetching when combined with multithreading, however they state that their prefetching method was meant for a single threaded processor and should be done differently for a multithreaded processor.

The **conditional-switch** model adds cacheing to the **explicit-switch** model. The code appears the same as that for the explicit-switch model: there is a group of load instructions, followed by a context switch instruction, followed by the instructions that use the loaded data. The difference is that the context switch instruction is treated as a *conditional* switch instruction. If any of the loads preceding the switch instruction missed in the cache, a context switch is performed as expected. But if all of the preceding loads hit, the context switch instruction is ignored and the thread continues executing. This model provides the same benefits of grouping and cacheing that were possible in the **switch-on-use-miss** model, but it may be simpler to implement.

In this paper we concentrate on the **switch-on-load**, **explicit-switch**, and **conditional-switch** models. This allows us to demonstrate the effectiveness of grouping in improving the performance of multithreading and in reducing the number of threads needed by each processor.

3 Multiprocessor Model

We have targeted this research at shared memory applications that use a fixed set of processes. This is typical for applications written in C or FORTRAN. The applications fork a set of processes and use that set for the entire computation by either statically or dynamically scheduling the work to be done.

For benchmarks, we use parallel programs written for the Sequent[19]. The Sequent allows shared storage to be allocated both statically (using a `shared` declaration) and dynamically (using `shmalloc()`). A potential problem arises because a pointer can point to either local or shared data. This would require dynamic address testing to determine whether or not an address is in shared memory. We avoid this difficulty by assuming that all memory references can be statically classified by the compiler as either local or shared. After looking at many programs, it is clear that this assumption is valid. Programmers write programs where they know which data is in shared memory. There are several techniques which could be used to determine this information. The simplest is to augment variable declarations to allow specifying which variables point to shared data. In our simulator we use trace analysis to determine this information.

For the CPU we assume a typical pipelined RISC processor, except that each thread has its own set of 32 integer and 32 floating-point registers. We use the instruction set and timings of the MIPS R3000[12], but we have supplemented it with a few additional instructions which are needed on a multiprocessor. We added `Load-Double` and `Store-Double` instructions to reduce the number of network messages, and we provided both local and shared versions of all load and store instructions. We added a `Fetch-and-Add` instruction as the synchronization primative and built higher level synchronization primatives such as locks and barriers out of `Fetch-and-Add`'s and spinning.

We assume that there is no overhead (in lost cycles) for context switching. This is consistent with switch-every-cycle machines[2, 11, 14, 17, 20] but in conflict with switch-on-miss machines[1, 9, 23, 25] which assume a cost of several cycles for clearing the processor pipeline. The key difference is the ability to recognize that an instruction will cause a context switch as that instruction enters the pipeline. This is possible for the switch-on-load and explicit-switch models studied in this paper since the context switch is implied by the opcode alone, and thus the context switch is identified while the instruction is in the decode stage of the cpu pipeline. At this point, the first instruction from the next thread to be executed should have already been prefetched. It can then be decoded on the next cycle and thus no cycles will be wasted.

We assume that remote accesses are returned in the same order in which they are sent, and, as a result, we use round-robin scheduling of the threads on a processor. Round-robin scheduling is optimal under ordered delivery since whenever the next thread in round-robin order must wait for an access to return, all other threads will be waiting for the return of accesses that were issued later. While some networks support ordered delivery of messages[22], most other large scale networks do not. Even on these networks round-robin scheduling may still be a good choice since it provides fairness in scheduling. Fairness is important for programs that use static load balancing since if one thread is scheduled less often than the others, that thread will finish later and delay the entire computation.

We expect that large shared memory multiprocessors will be built with multi-stage packed-switched networks such as the butterfly network used in the NYU Ultracomputer[8] and the IBM RP3[21], and that the network will support combining of messages. If hardware combining is not available, software combining techniques could be used for barriers[26]. We do not simulate the network, but instead assume that it has a constant 200 cycle round trip latency. We expect the average latency of a 1024 processor machine will fall in this range. Other researchers also expect latencies in the hundreds of cycles. The DASH project[9] has latencies of 90 cycles for a 16 processor machine when ignoring network congestion. With congestion, latencies will increase. Dally[7] found average round trip latencies of 144 cycles on a 64 processor butterfly when running at 50% utilization.

In a real network, access latencies depend on network loading, which is a function of both the network bandwidth and the bandwidth requirements of the application. There can also be a large variance in latency[7] because of congestion within the network. We have purposely chosen not to model a specific network because it would obscure the main focus of this research which is to evaluate the ability

Application	Lines	Cycles	Description & Problem size
sieve	242	106 M	counts primes < 4,000,000
blkmat	409	87 M	blocked matrix multiply — 200 × 200 matrices
sor	332	258 M	S.O.R. solver for Laplace's equation — 192 × 192 grid
ugray	10784	1353 M	ray tracing graphics renderer — gears (7169 faces), 20 × 512 slice of image
water	1368	1082 M	simulate a system of water molecules — 343 molecules, 2 iterations
locus	6347	665 M	route wires in a standard cell circuit — Primary2 (1290 cells × 20 channels)
mp3d	1510	192 M	simulate rarefied hypersonic flow — 100,000 particles, 10 iterations

Table 1: Parallel Applications

of multithreading to hide long memory latencies.

Caches are assumed for both instructions and local memory, but the effects of misses in these caches has been ignored. The instruction cache and local data cache are expected to have high hit rates just as on uniprocessors. Local data misses are serviced locally and would not cause a context switch. However instruction cache misses are serviced over the network and in a real machine, should cause a context switch. We believe these assumptions will not qualitatively affect the results.

3.1 Simulator

Simulating a 1024 processor machine requires enormous amounts of both time and memory. Far too much, in fact, to ever allow simulating large enough computations to keep such a machine busy. Since we are interested in the ability of multithreading to hide long memory latencies, we have reduced the number of processors simulated, but not the latency. We feel that this makes our latency tolerance results more directly applicable to a 1024 processor machine.

We have built a very fast simulator based on augmenting the object code of applications in a similar fashion to the `pixie`[15] profiling system for MIPS machines. With this code augmentation technique we are able to use optimized code as would be used on the actual machine. All applications were compiled at optimization level "-O2". Most instructions are directly executed by the host machine in a single cycle. A few additional cycles are used for loading and storing relevant registers, and counting execution cycles. On average, two instructions are added for each original instruction. Shared accesses are handled by the simulator, and require scheduling and context switching among many threads. These shared accesses take from 300 to 1000 cycles to simulate, but overall, the simulator is very fast since only a small fraction of instructions are shared accesses. Applications are typically simulated at speeds that range from 20 to 70 times slower than direct execution.

3.2 Applications

Table 1 lists the applications used in this study. `Sieve`, `blkmat`, and `sor` are toy applications developed by the authors. `Ugray` is from Berkeley[3]. And `water`, `locus`, and `mp3d` are from the Stanford SPLASH[24] benchmark set. We show the problem sizes and the number of cycles (in millions) that the applications would require on a single (0 latency) processor. It should be noted that these are much smaller than problems that would be solved on a real parallel machine. For example, a 100 mips workstation could solve the largest of these problems in under 14 seconds. Even at these small sizes, many of the applications already have sufficient parallelism for 1024 threads. Problems large enough for a 1024 processor parallel machine will likely have enough parallelism for hundreds of thousands of threads. More than enough for multithreading.

Figure 2 shows the performance of the seven applications on an ideal shared memory machine. This ideal machine has no contention and zero latency to shared memory. Such a machine would be impossible to build, but it corresponds to an upper bound on achievable performance. Rather than show the standard speedup curves, we have plotted the efficiency vs. the number of processors (efficiency = speedup / number of processors). For example, the `mp3d` program achieves a speedup of 778 when using 1024 processors, and thus has an efficiency of .76 = (778 / 1024). These curves are all computed with a fixed size problem. As the number of processors increases, the work is divided more finely and the efficiency drops. This lower efficiency at finer granularities results from uneven load balancing, synchronization overhead, and parallelization overhead. If, on the other hand, the problem sizes were increased, there would be more parallelism and thus more processors could be utilized.

Four of the applications (`sieve`, `blkmat`, `sor` and `mp3d`) have speedups of 700 or better with 1024 processors. Simulation time was too large for the other programs to allow simulating large enough problems to achieve these speedups. The `water` application stands out in the graph because of its erratic behavior. With 256 processors the efficiency is only .56, but with 343 processors the efficiency rises to .79 . This rise in performance results from the static load balancing used in the application. The load balancing works best when the number of processors divides evenly into the number of molecules (which is 343).

All programs studied in this paper have a single forked phase during which the parallel computation is performed. Any work done outside this forked phase has been excluded from the results presented in this paper. This excluded work typically consists of reading input files, performing initialization, and writing result files. There are a number of reasons for excluding this serial computation. First, as problem sizes are increased, the startup code becomes a smaller and smaller fraction of the total computation[10]. Second, many of these applications were written for today's small shared memory machines. Often much of the initialization could have been parallelized, but this wasn't deemed necessary for a small machine. Third, we expect that large parallel machines will do their input and output in parallel.

4 Switch-On-Load

In this section we evaluate the switch-on-load multithreading model. This is a simple model that works well for some applications but poorly for others. It serves as a baseline for comparison and helps motivate more sophisticated multithreading models.

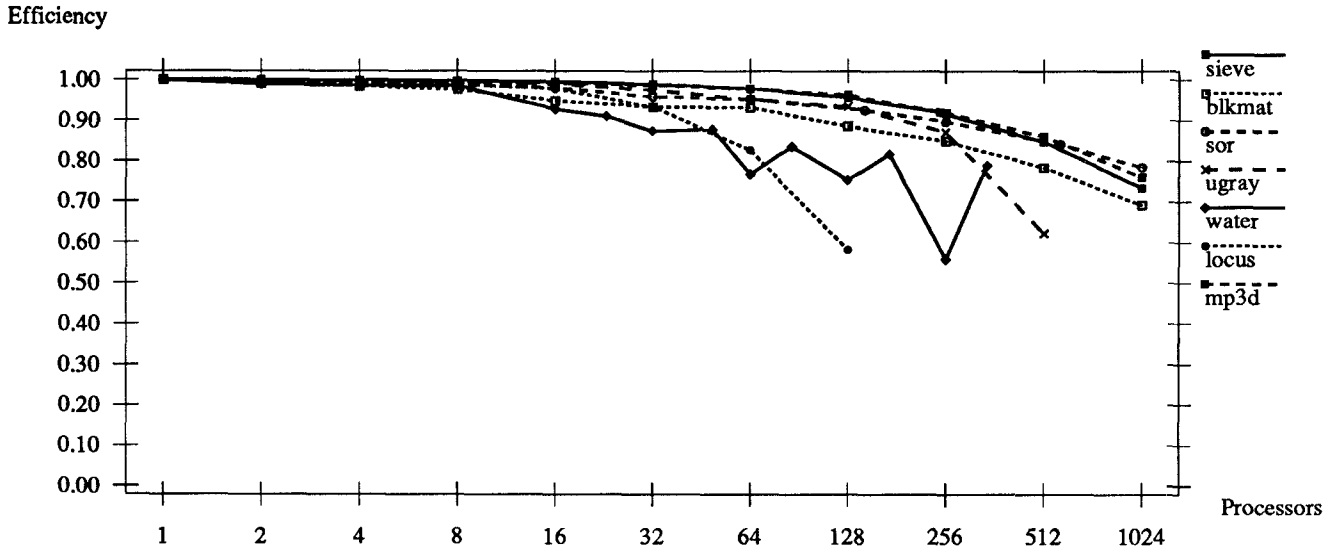


Figure 2: Efficiency on ideal (0 latency) machine.

App.	Distribution 1–9	10–19	20–29	30–39	40–49	50+	mean
sieve	—	95%	5%	—	—	—	19
blkmat	—	31%	1%	2%	3%	63%	137
sor	79%	19%	—	1%	2%	—	6
ugray	65%	10%	3%	11%	7%	4%	18
water	83%	8%	6%	—	—	4%	44
locus	93%	4%	2%	1%	—	1%	7
mp3d	81%	6%	5%	5%	—	3%	9

Table 2: Switch-On-Load: Run-Lengths Between Shared Loads

4.1 Run-Lengths Between Shared Loads

Table 2 shows the average run-lengths for the applications studied. Run-length is defined as the number of cycles between context switches. For example, if a shared load instruction occurs every 10 cycles, the run-length would be 10 since a context switch is done on every shared load. The mean run-length can be used to estimate the number of threads that will be need to hide the memory latency. For example, with a mean run-length of 10 cycles, 20 threads could hide the 200 cycle latency. Counting the thread whose latency is being hidden, a multithreading level of 21 would be sufficient. We would prefer to have a small multithreading level because it requires less total threads and allows small problems to be solved more efficiently.

Using the mean run-length to estimate the multithreading performance fails when there is variation in the run-length distribution. Sometimes all of the threads will execute run-lengths longer than the mean, and the latency will be covered easily. But at other times, threads will execute run-lengths shorter than the mean, and the full latency will not be covered. In this situation, either a higher multithreading level should be used or else some cycles will be wasted waiting for memory.

For the seven applications studied, only `sieve` has a fairly constant run-length distribution. (It runs through a large array marking numbers as non-prime at a constant rate.) The other applications, have more complex behavior. There are usually many different calculations being performed. Some require only a few cycles, while others require hundreds. This leads to large variations in run-lengths. The `blkmat` application stands out because of the exceptionally high mean run-length. This occurs because it makes private copies of data to reduce the number of shared accesses. The applications `sor`, `locus` and `mp3d` have very short run-lengths and will need large multithreading levels to hide their latency.

4.2 Performance

Figure 3 shows the performance of the `sieve` application under different levels of multithreading. The top curve shows the performance on an ideal (0 latency) shared memory multiprocessor. The other curves shown are with 200 cycle latency. Without multithreading, the processors are busy only 9% of the time. As extra threads are added, more of the latency is covered, and the efficiency rises, until at a multithreading level of 12, nearly 100% efficiency is achieved.

On the ideal machine, the efficiency of this application starts to drop when more than 100 processors are used. This drop in performance arises because of imperfect load balancing and the overheads of parallelization and synchronization. Under multithreading, the drop in performance occurs at the same point (100 threads), but because there are several threads assigned to each processor, the drop off in performance occurs at a proportionately lower number of processors. With larger problem sizes we expect that the region of linear speedup will extend further to the right. And thus with large enough problems for a 1024 processor machine, we expect to see the same multithreading behavior at 1024 processors that is observed here using a lesser number of processors.

The performance of the other applications is summarized

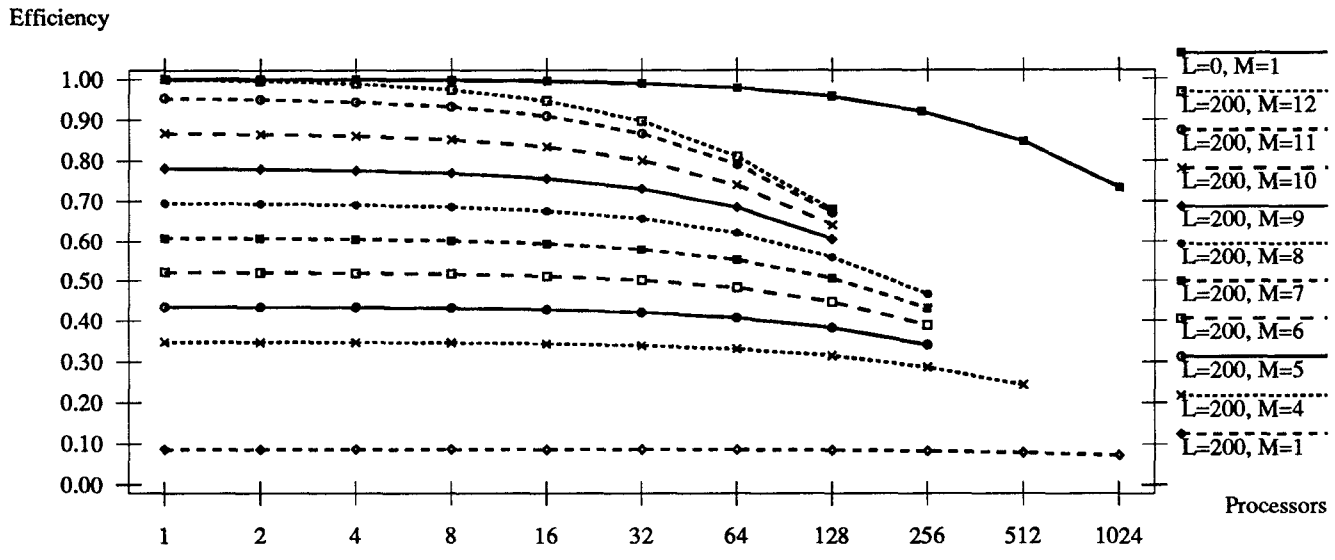


Figure 3: Multithreading Efficiency for sieve (L = Latency, M = Multithreading level)

Application (processors)	Desired Efficiency 50%	60%	70%	80%	90%
sieve (16)	6	7	9	10	11
blkmat (32)	1	2	2	4	—
sor (8)	28	40	—	—	—
ugray (8)	9	12	—	—	—
water (10)	5	7	10	12	—
locus (2)	17	21	25	32	—
mp3d (8)	13	15	19	22	28

Table 3: Switch-On-Load: Multithreading needed to achieve % efficiency.

in table 3. The multithreading levels needed to achieve 50, 60, 70, 80 and 90% efficiency are shown. Most of the applications could not achieve all of these efficiency levels. As the multithreading is increased, more total threads are used and the applications enter the domain where the problem sizes are too small for the number of threads. For the `sieve` application table 3 shows the results when using 16 processors. With a multithreading level of 11, the program uses 176 threads and runs at 90% efficiency. Note that in figure 3 we could have chosen any point from 1 to 16 processors and the multithreading level needed to achieve 90% efficiency would have remained the same. Similarly for each of the applications, we have reported the performance at a point just before the performance drops off.

While some of the applications achieve high efficiencies, other applications such as `sor` and `ugray` reach an upper bound around 60% efficiency. The reason for this bound involves the run-length distributions presented in table 2. With the `sor` application, for example, 39% of the run-lengths are 1 cycle and another 39% are 2 cycles. These short run-lengths contribute little towards hiding latency and it is inevitable that cycles are lost waiting for memory.

5 Explicit-Switch

We can do better than the switch-on-load model by looking at the data dependency structure of the applications. Most programs do not load a single value and then perform a calculation with it. Instead, they usually load several values and compare or combine them in a calculation. The processor should issue the group of loads together and then context switch only once, rather than context switching after every load.

5.1 Compiler Optimization

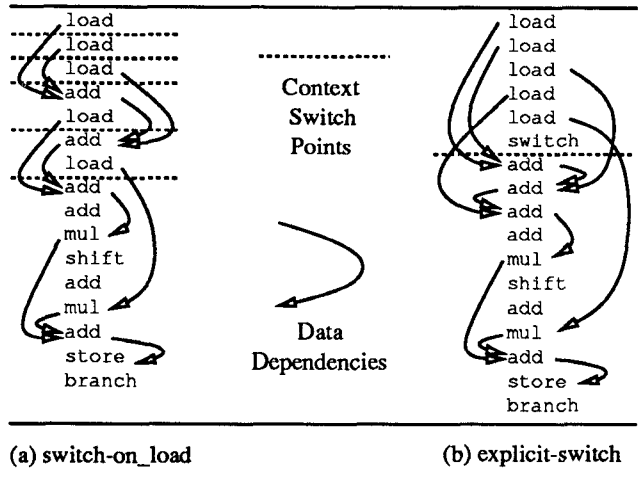
The grouping together of loads improves the performance of multithreading in two ways. First, by eliminating context switches, it increases the run-length (number of cycles) between context switches. With a longer run-length, less threads are needed to cover the memory latency. Second, it improves the distribution of run-lengths by eliminating many of the small run-lengths that result from back to back loads.

The inner loop of the `sor` application is shown in figure 4(a) as an example. Without grouping, the 5 loads are issued one at a time, with a context switch after each one. In figure 4(b) the code has been reorganized so that all 5 loads are grouped together and are then followed by a single context switch instruction. Rather than having four short run-lengths followed by one long run-length, there is now just a single long run-length.

A compiler designed for a multithreaded architecture will group shared loads whenever possible. Since the compilers we have today don't do this grouping, we wrote a post-processor which finds the basic blocks in an object file, does dependency analysis within the basic blocks, and then reorganizes the instructions so as to group shared loads together. It then inserts a single context switch instruction after each group of independent shared loads. Because we do this analysis at the assembly language level, we must

Application	Distribution 1–9	10–19	20–29	30–39	40–49	50+	mean	loads / switch
sieve	—	63%	37%	—	—	—	19	1.0
blkmat	—	23%	9%	2%	3%	63%	138	1.0
sor	—	—	80%	2%	17%	—	30	4.7
ugray	50%	17%	5%	15%	2%	11%	23	1.3
water	—	36%	9%	22%	13%	19%	210	4.8
locus	93%	3%	2%	1%	—	1%	8	1.05
mp3d	23%	23%	28%	19%	—	7%	23	2.3

Table 4: Explicit-Switch: Run-Lengths Between Context Switches



(a) switch-on_load (b) explicit-switch

Figure 4: Inner loop of sor.

make pessimistic assumptions[1] which restrict our ability to reorganize the code. Nevertheless our code reorganization works very well for basic blocks. Compiler based optimization could do even better by looking beyond the scope of a single basic block.

Table 4 shows the new run-length distributions for the applications. The last column shows the level of grouping achieved. The sor and water applications benefited the most. In the inner loop of sor, for example, 5 loads were grouped together between context switches. Where previously their run-length distributions were dominated by short run-lengths (of 1 or 2 cycles), now those short run-lengths have been completely eliminated. In contrast, the locus and ugray applications did not have much improvement. These applications have very short basic blocks, and grouping is needed beyond the scope of a single block. Two of the applications, sieve and blkmat, did not benefit at all from grouping, but this is unimportant since they access shared memory one item at a time and already have good run-length behavior.

Table 5 shows the improved performance results with the explicit-switch multithreading model. The applications now achieve higher performance and use fewer threads. For locus the mean run-length of 8 cycles is still too short to allow a small number of threads to cover the latency, but for the other applications, 14 or fewer threads are now sufficient to maximize performance.

One penalty of having explicit switch instructions is that they take a cycle that might have otherwise been used for computation. A further penalty arises from the code reorganization. The original code is optimized so that the execution of floating point instructions is overlapped with that of the load instructions. When we reoptimize the code to group loads together, we often add a few cycles to the execution time because operations are no longer overlapped as tightly. The last column of table 5 shows the performance penalty due to the code reorganization and added context switch instructions. This penalty is often just a few percent, and in all cases it is overshadowed by the benefits of grouping.

[1] We assume that every shared store might have a conflict with every shared load because of address aliasing.

Application (procs)	Desired Efficiency 50%	60%	70%	80%	90%	grouping overhead
sieve (16)	6	7	9	10	11	2%
blkmat (32)	1	2	2	4	—	0%
sor (16)	5	6	8	9	—	10%
ugray (8)	6	8	11	—	—	2%
water (20)	1	2	3	4	6	0%
locus (2)	16	20	26	—	—	11%
mp3d (16)	6	7	8	10	14	4%

Table 5: Explicit-Switch: Multithreading needed to achieve % efficiency. (Efficiency measurements account for grouping overhead.)

5.2 Grouping Beyond Basic Blocks

For the locus and ugray applications we were disappointed that grouping did not perform better. After examing these programs, we found that both programs could benefit from inter-block grouping. We observed that the inter-block grouping opportunities often involved accesses to several fields in a small structure. These accesses were split among several basic blocks because of condition tests, but in fact could have all been issued before the condition tests were done.

Since we don't have a compiler which does inter-block grouping, we designed a simple experiment to estimate the potential grouping available. We simulate a very small cache associated with each thread. The cache has a line size of 32 words, but only one line. We assume that any loads which hit in this cache are in the same structure or array as the preceding reference and thus could have been grouped.

For ugray 42% of the loads hit in this cache, and the grouping factor increased from 1.3 to 1.9 . For locus the hit rate is 84%, and the grouping factor increased from 1.05 to 6.6 . This dramatically shows the potential for compiler based grouping. Table 6 shows the revised multithreading figures based on the additional grouping predicted by this experiment.

Application	explicit-switch cycles / access	explicit-switch bandwidth bits/cycle	conditional-switch hit rate	conditional-switch cycles / access	conditional-switch bandwidth bits/cycle
sieve	13.7	9.3	99%	3333	0.1
blkmat	114	1.4	65%	326	1.2
sor	5.3	30.2	98%	246	2.3
ugray	24.1	5.8	92%	228	1.3
water	44.1	3.4	93%	518	0.7
locus	8.3	15.4	91%	146	2.6
mp3d	6.4	20.2	72%	23	18.7

Table 7: Comparison of bandwidth requirement under explicit-switch and conditional-switch multithreading models.

Application (procs)	Desired Efficiency 50%	60%	70%	80%	90%	loads / switch
sieve (64)	2	2	3	5	—	11.3
blkmat (64)	1	1	2	3	—	1.4
sor (16)	5	6	8	9	—	5.0
ugray (12)	5	6	8	—	—	1.9
water (20)	1	2	2	3	5	8.1
locus (8)	4	5	6	7	—	6.6
mp3d (32)	4	5	6	8	11	3.3

Table 6: Explicit-Switch: estimated performance with interblock grouping.

This experiment is only a means to estimate the available grouping opportunities. To verify the results, we examined the code of **ugray** and `locus` to see where the cache hits were comming from. In **ugray** we found that 47% of the identified grouping opportunities were valid. These were cases where one field in a structure was examined, and if it met some condition, a second field was loaded and used in a computation. A smart compiler could group these two loads together by speculatively loading the second value in the expectation that the test would succeed and thus the second load would be needed. The other 53% of the grouping opportunities identified by the cache experiment turned out to be cases of coincidental memory allocation such as when two structures were allocated in the same cache line. While examing the code, however, we also found many interblock grouping opportunities that were missed by the cache. Overall, it remains unclear whether this experiment overestimated or underestimated the interblock grouping potential for **ugray**.

For the `locus` application, a single instruction was found to be responsible for 95% of the cache hits. This turned out to be in a loop that was stepping horizontally though a large two dimensional array. A compiler could easily group these loads by unrolling the loop. In addition, we found similar loops that stepped vertically through the array. The same compiler unrolling technique could group these loads as well, but they were missed by the cache. Thus for `locus` our experiment underestimated the potential for interblock grouping.

Research in compiler optimization is needed to fully explore the idea of grouping accesses. We have shown that grouping within basic blocks is simple and effective for many application, and that further grouping is available by looking beyond basic blocks. Much of this interblock grouping will come from compiler techniques such as unrolling loops and speculative loads. The speculative loads will need careful consideration since they will increase grouping at the cost of increased network bandwidth.

6 Conditional-Switch

The biggest problem with the explicit-switch model is that it requires a large bandwidth to shared memory. In this section we present preliminary results on the effects of adding a cache to a multithreaded processor. The cache is able to reduce the memory bandwidth requirement by filtering out a large fraction of the accesses. To be cost effective, however, the increased cost and complexity of adding caches must be more than compensated for by the reduced cost of the network.

6.1 Bandwidth

For this preliminary study we have chosen to look at small caches that could be included as part of a single chip processor. We experimented with several cache sizes and line sizes and found that a 64 K-Byte, write back cache, with a 16 byte line size and 4 way set associativity is a reasonable choice. Selection of an optimal cache configuration is beyond the scope of this paper.

Coherency is maintained with the Censier and Feautrier[4] invalidation based cache coherency protocol. This protocol maintains a full directory of owners for each data line. It serves as a convenient starting point for comparison with other research, however we expect that this may not be the most cost effective choice because of its large hardware cost[18].

Table 7 shows the applications' bandwidth requirements under the explicit-switch model compared to those under the conditional-switch model. We have shown both the access rate, which is shown in terms of the average number of cycles between accesses, and the bandwidth requirement which is shown in bits per cycle. The bandwidth figure includes the overhead of message headers and result, acknowledgement, and invalidation messages.[2] The bandwidth does not decrease proportionally to the access rate because of the extra coherency overhead and the larger message sizes needed to load a full line of data.

The cache works well for all of the applications except `mp3d`. The hit rates are above 90% and the bandwidth requirements are reduced to well under 4.0 bits per cycle. This bandwidth figure is the sum of both the forward and return traffic, and thus it suggests that a network with channels as narrow as 2 bits in each direction would have sufficient bandwidth to support these application. In reality the channels might need to be wider than this because traffic will be bursty and have periods of higher bandwidth requirements.

[2] We have excluded messages used in spinning on locks and barriers, since we expect a real machine to provide mechanisms to perform these operations without spinning.

Application (processors)	Desired Efficiency 50%	60%	70%	80%	90%
sieve (256)	1	1	1	1	—
blkmat (64)	1	1	2	3	—
sor (16)	1	1	3	4	—
ugray (32)	1	1	2	2	4
water (29)	1	1	1	2	3
locus (10)	1	2	2	4	—
mp3d (32)	3	4	5	6	9

Table 8: Conditional-Switch: Multithreading needed to achieve % efficiency.

Simulations using realistic networks are needed to fully explore this issue.

The mp3d application has very poor reference locality and thus benefits little from cacheing. To support this application as is, the network will have to supply much higher bandwidth. We would be interested in seeing if this application could be rewritten to improve is locality.

6.2 Multithreading Level

An additional benefit of cacheing is that it can further reduce the level of multithreading needed to achieve high performance. Since many context switches are not taken, the run-lengths between taken context switches increases, and this allows hiding the memory latency using fewer threads.

Skipping too many context switches, however, can causes problems. In the ugray application, for example, we found that long sequences of cache hits allowed threads to have some run-lengths that lasted for thousands of cycles. These long run-lengths caused problems when another thread on the same processor was in a critical region. The thread in the critical region would like to complete and exit the critical region as quickly as possible, but when the thread context switched, the other threads on the processor would sometimes execute for a very long time before the first thread was resumed and allowed to finish. The result was that locks ended up being held for much longer than they needed to be, and the increased lock contention hurt performance.

The simplest solution to this problem is to limit the interval between context switches. For this simulation we set a flag after a thread has executed for 200 cycles. When this flag is set, the next context switch instruction will cause a context switch regardless of whether there were any preceding cache misses. This mechanism is adequate for this study, but there is room for improvement by using more sophisticated scheduling policies such as priority scheduling of threads inside critical regions.

Table 8 shows the performance of the application under the conditional-switch multithreading model. Under this model execution efficiencies of 80% or better can be achieved with 6 threads or less. This small number of threads is important because it reduces the size of the register files need to support multithreading.

The conditional-switch multithreading model cannot always context switch in a single cycle as the switch-on-load and explicit-switch models could. The context switch decision is based on whether or not any of the preceding load instructions miss in the cache. If a miss occurs several cycles before the context switch instruction enters the CPU pipeline, the context switch will be correctly predicted, but if the outcome of a load is not known when the context switch instruction enters the pipeline, a few cycles may be wasted. For the applications studied here, this effect was minor.

The results from table 8 show that many of the applications were able to achieve efficiencies of 50 or 60% with just a single thread per processor. This is substantially better than previous results such as those from the Dash project[9] which reported processor utilizations ranging from 20% to 28% when using just a single thread per processor. The difference between their processor model and ours is that we are able to group references together because of the context switch instruction. This grouping lets us issue multiple accesses into the memory network before we wait for them to return. Our improved performance suggests that the grouping of shared accesses is an important capability that extends beyond multithreaded processors.

7 Conclusions

While previous studies of multithreading have shown mixed results, we study a broad set of applications and show that all of them can perform well under multithreading. The key idea introduced is to take advantage of the data dependency structure of the code and group shared loads together so that processors can avoid unnecessary context switches. This grouping is facilitated by the introduction of an explicit context switch instruction and compiler optimization techniques. Grouping increases the average run-length between context switches and also eliminates most of the troublesome short run-lengths from the run-length distributions.

It is difficult to compare results with other studies since different applications are studied, different problems sizes are used, and different latencies are assumed. One common point for comparison is possible with the DASH project. Gupta and Hennessy[9] studied the mp3d application under the switch-on-miss multithreading model. They reported an efficiency of 50% with a multithreading level of 4. The explicit-switch model studied here achieves similar efficiency at this multithreading level while tolerating a latency more than twice that used in the DASH study. The improved latency tolerance of the explicit-switch model shows the importance of grouping.

We have introduced a classification scheme for multithreading models and have started the exploration of this design space. With the explicit-switch model, 70% or better efficiency can be achieved without any cacheing of shared memory, and requiring only a moderate level of multithreading. The only difficulty is that the network bandwidth requirements are high for some applications. Cacheing together with the conditional-switch model addresses this problem and for most applications is able to reduce the bandwidth to acceptable levels.

References

[1] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *The 17th Annual Int. Symp. on Computer Architecture*, pages 104–114, 1990.

[2] Robert Alverson, David Callahan, Daniel Cummings, et al. The TERA Computer System. In *1990 Int. Conf. on Supercomputing*, pages 1–6, 1990.

[3] Bob Boothe. Multiprocessor Strategies for Ray-Tracing. Master's thesis, U.C. Berkeley, September 1989. Report No. UCB/CSD 89/534.

[4] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[5] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache Coherency in Large-Scale Multiprocessors. *IEEE Computer*, 23(6):49–59, June 1990.

[6] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controled Threaded Abstract Machine. In *ASPLOS-IV Proceedings*, pages 164–175, 1991.

[7] William J. Dally. Virtual-Channel Flow Control. In *The 17th Annual Int. Symp. on Computer Architecture*, pages 60–68, 1990.

[8] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer — Designing a MIMD, Shared-Memory Parallel Machine. In *Conf. Proc. of the 9th Annual Symposium on Computer Architecture*, pages 27–42, 1982.

[9] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *The 18th Annual Int. Symp. on Computer Architecture*, pages 254–263, 1991.

[10] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, pages 532–533, May 1988. Technical Note.

[11] Robert H. Halstead, Jr. and Tetsuya Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *The 15th Annual Int. Symp. on Computer Architecture*, pages 443–451, 1988.

[12] Gerry Kane. *MIPS RISC Architecture*. Prentice Hall, 1989.

[13] Alexander C. Klaiber and Henry M. Levy. An Architecture for Software-Controlled Data Prefetching. In *The 18th Annual Int. Symp. on Computer Architecture*, pages 43–53, 1991.

[14] Janusz S. Kowalik, editor. *Parallel MIMD computation : the HEP supercomputer and its applications*. MIT Press, 1985.

[15] MIPS Computer Systems. *MIPS language programmer's guide*, 1986.

[16] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A Killer Micro for A Brave New World. Technical report, MIT Lab. for Comp. Sci., 1991. Computation Structures Group Memo 325.

[17] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *The 16th Annual Int. Symp. on Computer Architecture*, pages 262–272, 1989.

[18] Brian W. O'Krafka and A. Richard Newton. An Empirical Evaluation of Two Memory-Efficient Directory Methods. In *The 17th Annual Int. Symp. on Computer Architecture*, pages 138–147, 1990.

[19] Anita Osterhaug, editor. *Guide to Parallel Programming on Sequent Computer Systems*. Prentice Hall, 1989.

[20] Gregory M. Papadopoulos and David E. Culler. Monsoon: an Explicit Token-Store Architecture. In *The 17th Annual Int. Symp. on Computer Architecture*, pages 82–91, 1990.

[21] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proc. 1985 Int. Parallel Processing Conf*, pages 764–771, 1985.

[22] Abhiram Gorakhanath Ranade. *Fluent Parallel Computation*. PhD thesis, Yale University, May 1989.

[23] Rafael H. Saavedra-Barrera, David E. Culler, and Thorsten von Eicken. Analysis of Multithreaded Architectures for Parallel Computing. In *ACM Symp. Paral. Alg. Arch.*, July 1990.

[24] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical report, Computer Systems Laboratory, Stanford, 1991. Tech. Rpt. #CSL-TR-91-469.

[25] Wolf-Dietrich Weber and Anoop Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *The 16th Annual Int. Symp. on Computer Architecture*, 1989.

[26] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing Hot-spot Addressing in Large-Scale Multiprocessors. In *Int. Conf. on Parallel Processing*, pages 51–58, 1986.