the hardware mechanisms necessary to support the multithreading models studied in this dissertation

Our goal is to show that the hardware is simple enough for a single chip implementation. In fact the MIT Alewife project is using a slightly modified SPARC processor as a multithreaded processor by using its register windows to hold multiple contexts instead[ALKK90]. We will present the ideas in this chapter as modifications or additions to an ordinary pipelined RISC processor, and we expect that these ideas can be incorporated into existing processor designs. These ideas should also extend to superpipelined and superscaler processors and thus the considerable development effort into these processors can be leveraged in the development of multithreaded processors.

8.1 Hardware for Explicit-Switch

This section describes the hardware mechanisms needed on a multithreaded processor that provides the **explicit-switch** multithreading model. We focus on **explicit-switch** because this was shown in Chapter 4 to be superior to **switch-on-load**. The hardware support required for **switch-on-load** is virtually identical.

Figure 8.1 shows a simplified data path for a RISC processor. A real processor would be more complex, but this diagram is sufficient for explaining the ideas developed in this section.

Figure 8.2 shows a revised data path that has been modified to support the explicit-switch multithreading model. This diagram shows support for a multithreading level of at most 4 threads per processor. There are 4 register sets, 4 program counters, and 4 instruction registers. Earlier simulations suggested that an actual processor should be built to allow perhaps 10 threads per processor if a latency of 200 cycles is expected.

In addition to the extra register sets, PC's, and IR's, we have shown the addition of a scheduler block, a larger control unit because of the increased complexity, and a path from memory directly into the register file. Each of these changes will be discussed below.

8.1.1 Pipelined Context Switch

While **switch-every-cycle** machines have no lost cycles due to a context switch, most research on other multithreading models has assumed non-zero context switch times. For example, Gupta *et al.*[GHG+91] assumed a cost of 4 cycles for flushing the processor

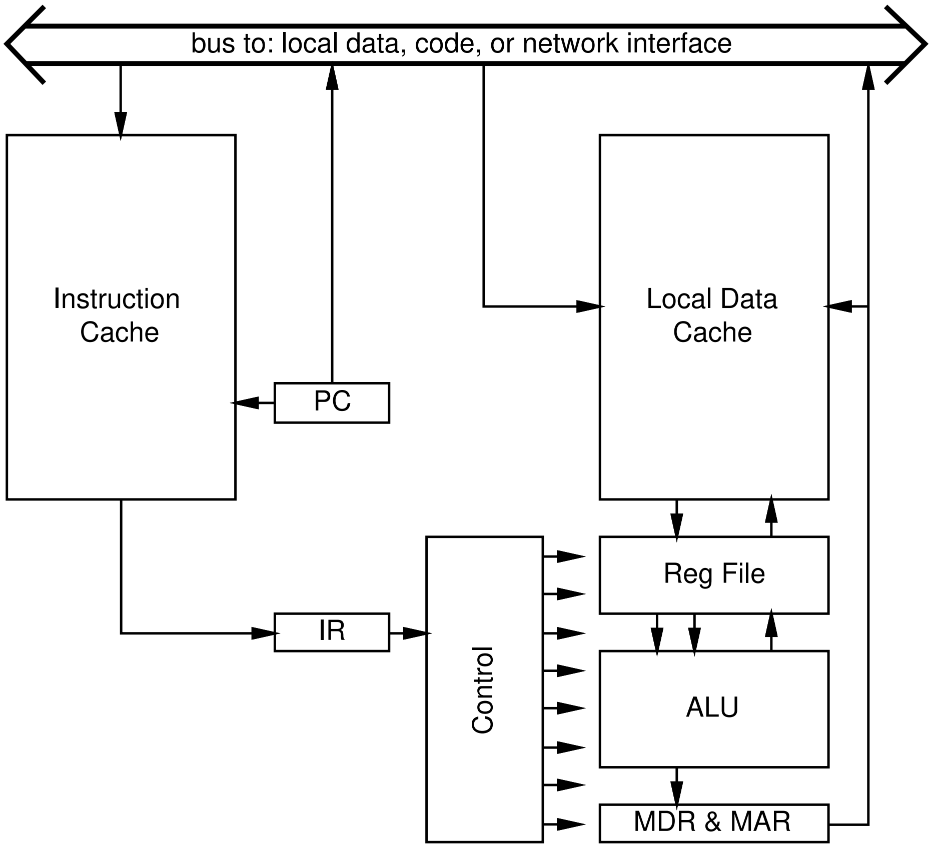

Figure 8.1: Datapath for RISC processor.

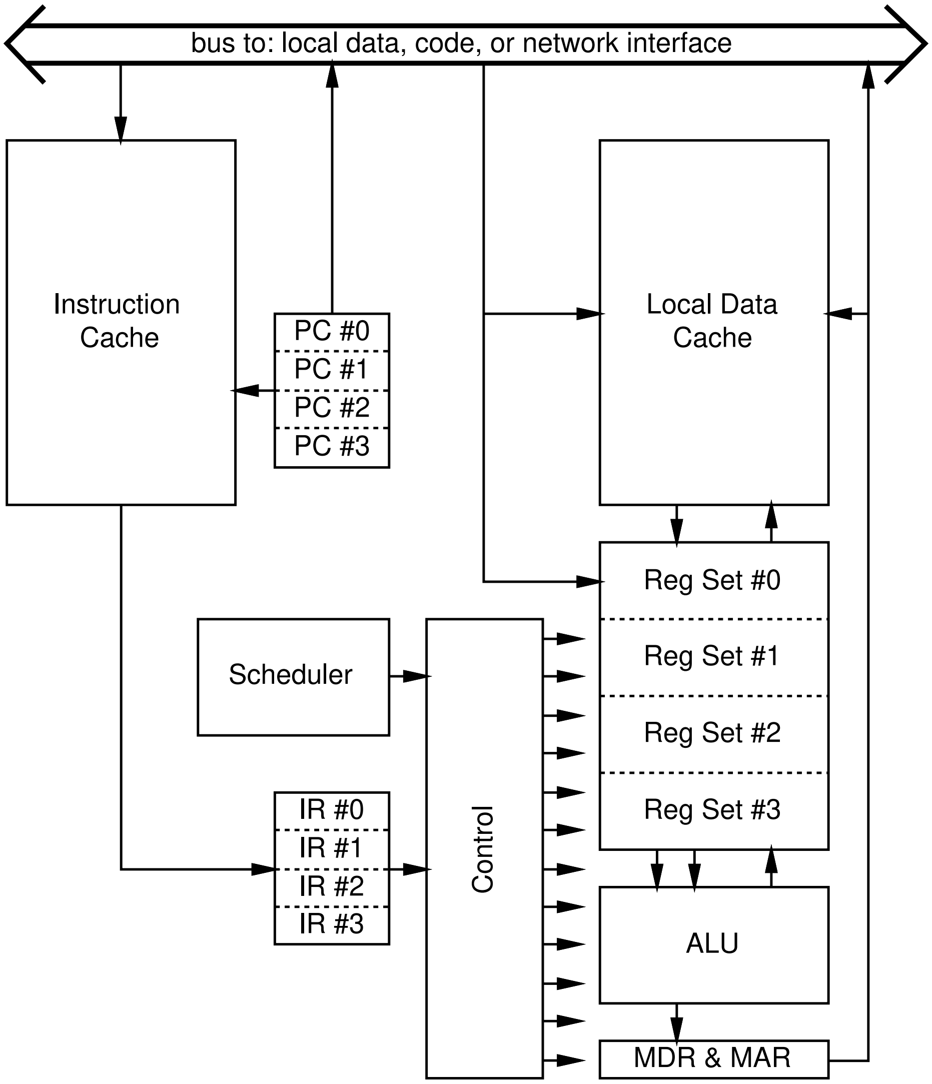

Figure 8.2: Datapath with changes for explicit-switch multithreading.

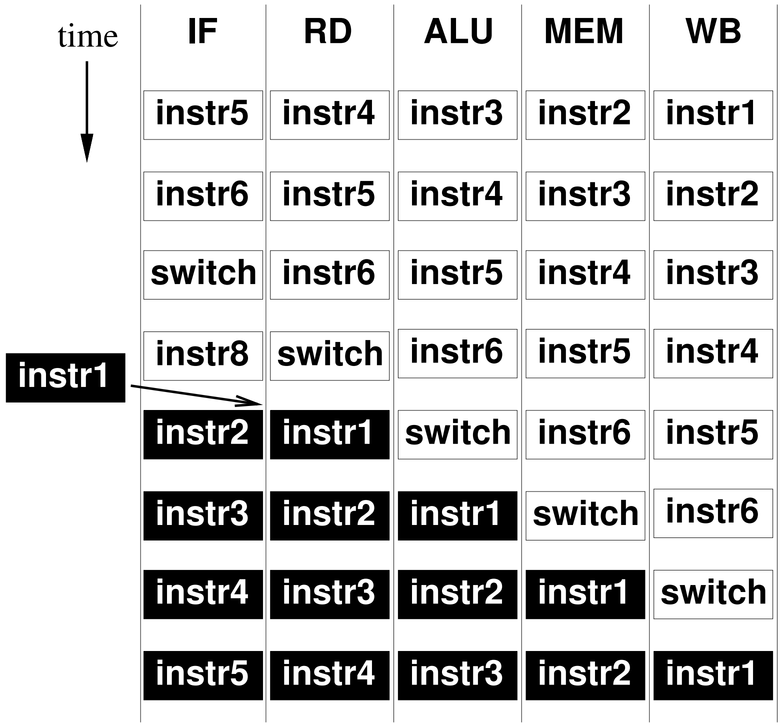

Figure 8.3: Pipelined context switch for explicit-switch.

pipeline, or 16 cycles for a less aggressive implementation. For the **explicit-switch** model considered in this section, we will show that flushing the processor pipeline is unnecessary and thus a single cycle context switch is possible.

The key to the single cycle context switch is to pipeline the context switch just as other operations are pipelined. Figure 8.3 shows the content of the processor pipeline[1] as it progresses from running one thread (shown in white), through a context switch, and the start of the next thread (shown in black). Each horizontal line represents a cycle. Time progresses down the page one cycle at a time, and the progress of instructions through the pipeline is shown by their movement to subsequent pipeline stages.

The seventh instruction for the white thread is a switch instruction. When the switch instruction is in the the decode stage (RD), the processor detects that the thread is to be context switched. At this point the processor has already fetched instruction eight of the white thread. However, instead of passing instruction eight through the pipeline, the processor can substitute the first instruction from the black thread if it is available. This is

[1]The pipeline stages are those of the MIPS R3000[Kan89].

Application (multithreading)	Efficiency switch = 1 cycle	Efficiency switch = 9 cycles	Loss
sieve (mt = 5)	86	86	0%
blkmat (mt = 3)	80	77	3%
sor (mt = 9)	83	67	16%
ugray (mt = 10)	76	68	8%
water (mt = 5)	90	89	1%
locus (mt = 8)	80	72	8%
mp3d (mt = 11)	89	73	16%
barnes (mt = 5)	87	83	4%

Table 8.1: Explicit-Switch: Performance loss with 9 cycle context switch.

the purpose of the additional instruction registers in Figure 8.2: to hold the prefetched first instruction from each thread. The multiple program counters are needed to allow immediate commencement of instruction fetching from the new thread.

The cost of the context switch for this method is just one cycle, which is the switch instruction itself. Notice that during the transition from the white thread to the black thread, instructions from both threads occupy the pipeline concurrently. This means that the black thread will start reading results from the black register set while the white thread is still writing results into white register set. To allow different pipeline stages to access different register sets, the processor must pass the register set number down the pipeline with the instructions. This transition between threads is similar to the transition between register windows in the SPARC processor.

The importance of a fast context switch depends upon the frequency of context switching. If context switches occur at intervals of hundreds of thousands of cycles, as they do in uniprocessor operating systems, the context switch time is not critical. But if context switches occur every 20 cycles, as they might on a multithreaded processor, even just a single cycle spent context switching decreases performance by 5%.

A slower context switch would result if the CPU pipeline were drained before starting a new thread[GHG$^+$91]. For a pessimistic value we assume an 8 cycle pipeline as on the MIPS R4000[MIP91], plus 1 cycle for the switch instruction. This gives a context switch cost of 9 cycles.

Table 8.1 shows the performance loss that would result from a 9 cycle context switch instead of a 1 cycle switch as has been studied in this dissertation. Simulation parameters are the same as they were in the studies of **explicit-switch** with inter-block grouping (refer to Table 4.4) except that the context switch time has been increased. The performance loss varies by application. The applications that have long average run-lengths (such as `sieve`, `blkmat`, `water`, and `barnes`) lose just a few percent of their performance. The other applications (`sor`, `ugray`, `locus`, and `mp3d`) have shorter run lengths, context switch more frequently, and thus incur a larger performance loss from the slow context switch. The worst performance loss is 16% which occurs for `sor` and `mp3d`.

The performance losses are lower than we expected for two reasons. First, average run-lengths for the explicit-switch model with inter-block grouping (as seen in Section 4.3.2) range from 30 cycles to more than 200 cycles. With longer run-lengths, the context switch time has less impact on performance. Second, some of the cycles lost to the slower context switch would have otherwise been lost to memory latency. This is a small effect, since multithreading is usually effective at hiding most of the memory latency.

This section has shown that pipelining the context switch is feasible and can provide as much as 16% performance improvement over a slower context switch that waits for the pipeline to drain before starting the next thread.

8.1.2 Result Matching

Multithreading allows issuing multiple memory references into the network to hide network latency. A difficulty arises because most networks do not preserve message order and thus the responses must be matched with the requests.

A simple solution is to send a small tag along with each message. This tag is later used to identify the returning result message. The tag should contain the thread number of the issuing thread and the register in which to put the result. This allows writing the result directly into the register file through a second write port at shown in Figure 8.2. By storing results directly into the register file, no special storage is needed and they are immediately available upon resumption of the thread. Alternatively, the second write port can be eliminated if the results are buffered and later written into the register file during cycles in which the processor does not write to the register file.

8.1.3 Scheduling

The main task of the scheduler is to determine when a thread is ready. In the multithreading models we have looked at, a thread becomes ready when all of its shared memory accesses have returned from the network. To keep track of outstanding references the scheduler will need a counter for each thread. The counter is incremented on each shared load issued into the network and is decremented upon its return. When the counter reaches zero, the thread becomes ready.

The ready threads may be scheduled with any sort of scheduling policy. For **explicit-switch** we used first-come-first-serve (which is the same as round robin when accesses return in order.) This is simple and fair. Other policies, such as those studied in Section 5.2.1, might provide some additional benefit, for instance, by causing timeouts on long run-lengths. However since long run-lengths are uncommon under **explicit-switch**, we expect the benefits of more complex policies will be small.

8.1.4 Multiple Register Sets

The largest change to the processor design in terms of chip area is the addition of multiple register sets. These multiple register sets are essential for fast context switching because without them it would take at least 128 cycles to save the register set[2] for the current thread out to memory and then load in the register set for the next thread. This overhead for context switching would overshadow any gains made from hiding the memory latency.

In a typical RISC processor design, the register file only occupies a few percent of the chip area. On the Stanford MIPS processor[PGH+84], for example, the register file occupied 8.3% of the chip area. For more recent processors with large on chip caches, the percentage of chip area used for the register file is even less. At this size, providing 10 register sets on chip, as was found to be sufficient to support the **explicit-switch** multithreading model, is conceivable. However, multithreading designs that allow hundreds of threads per processor, such as TERA. have so far been prohibited from considering single chip implementation because of the size of their register files. Other multithreading designs, such Monsoon and $\star$T[NPA92], do not provide a separate register set for each thread.

The precedent for increasing the register file size has already been set by the SPARC[Fuj88] chip and the Am29000[Man92]. SPARC has 120 integer registers, and the Am29000 has 192. Most of these are used to provide register windows to help speedup procedure call and return by shifting to a new register set rather than saving and restor-

[2] Here we assume that the register set is 32 general purpose and 32 floating registers.

ing registers to memory. However, the benefit of these register windows is small because compilers have been able to do a good job of avoiding most register saves and restores. Some researchers have therefore proposed using the register windows for multiple contexts instead[APRIL]. Unfortunately, as far as multithreading is concerned, the SPARC architecture does not provide register windows for the floating registers.

8.1.5 A Denser Register File

If the multithreading level and the number of register sets supported is small (say 4), the chip area used for the registers will be comparable to that used on the SPARC or AM29000 chips. Supporting $M = 4$ is thus clearly reasonable. However as the number of threads and register sets is increased, the chip area will become more of a concern and we therefore propose the following design which can be used for a denser implementation of the register file for a multithreaded processor.

The key to this design is that only the register set of the currently active thread is used by the processor. The other register sets sit idle until their thread is scheduled by the processor. Rather than keep all register sets in large multiported register cells, the inactive register sets can be kept in smaller single ported register sets until they are needed. If the single ported cells are implemented as dynamic memory, then on a VLSI chip they will require less than one twelfth the area of a regular multiported static cell[3].

The main obstacle in implementing this is being able to switch to a new active register set quickly at a context switch. At a context switch, the entire contents of the active register set must be saved into the inactive storage area, and the register set of the next thread must be loaded. For a register file of 32 registers, each of which is 32 bits, this constitutes 1024 bits that must be saved and another 1024 bits to be loaded. If done quickly, i.e. in parallel, this can be done in two cycles. In the first cycle all 1024 bits are transferred out of the active register file into the inactive register file, and in the second cycle the new 1024 bits are transferred in. Moving 1024 bits into (or out of) the register file in a single cycle would require 1024 wires, and this would be unwieldy.

This obstacle can be overcome by interlacing the inactive register file within the active register file as shown in Figure 8.4. This shows a single block that can be used in an array of 32 by 32 blocks to implement the full collection of active and inactive register sets.

[3] A static cell with 2 read and 1 write ports can be implemented in $64\lambda \times 41\lambda$. A dynamic cell in $10.5\lambda \times 18\lambda$[Waw91].

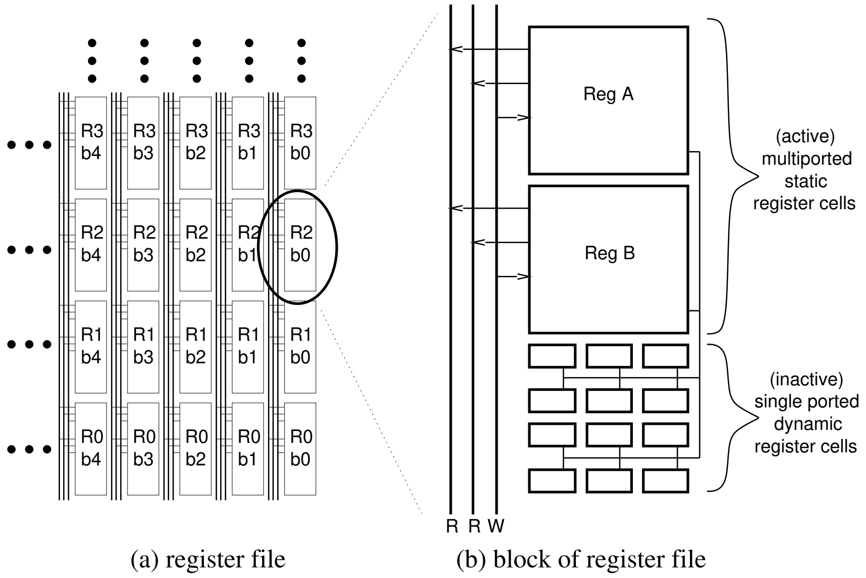

Figure 8.4: One bit of register file supporting 12 threads per processor.

The figure shows 12 single ported dynamic register cells for the inactive register sets and 2 multiported register cells for the active register sets. Usually just one of the two active register sets is used, except that when then processor is in transition from one thread to the next, instructions from both threads are in the pipeline and thus both register sets are needed.

The multiported register cells are used in an alternating fashion as shown in Figure 8.5. This example shows the transition from running one thread to the next and then to a third (shown in white, grey, and black respectively). At the start, the white thread has been executing out of the A-registers, and the B-registers have been loaded with the register set for the grey thread. When the white thread context switches, the registers for the grey thread are available and it can start executing immediately. The A-registers are retained until all of the white thread's instructions have exited the pipeline (except the switch instruction which does not use any registers). At this point, the A-registers are written into the dynamic memory storage area used for the inactive threads. The A-registers are now loaded with the register values of the black thread. By alternating between the A-registers and the B-registers, the active registers can always be kept available.

This technique allows enough registers for twelve contexts to be implemented in the space that would normally be needed for three. This compact register file keeps the bus

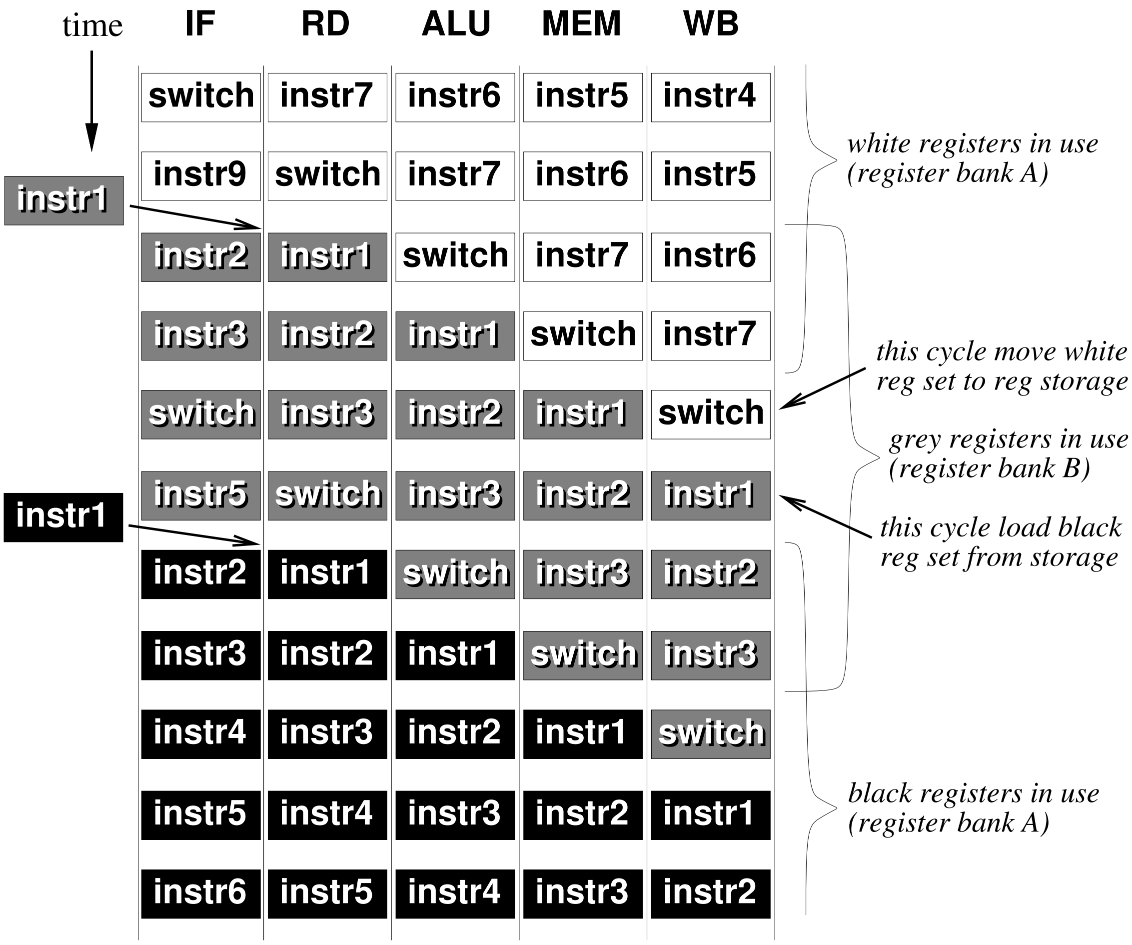

Figure 8.5: This shows the operation of the denser register file design. During the transition between threads, both the A register bank and the B register bank are in use simultaneously.

loading and lengths of the register read and writes busses within acceptable bounds. As the number of register sets is increased, this design becomes more desirable because additional register sets can be added using only one twelfth the area that would be required for adding a multiported static register set.

A potential complication arises because dynamic registers are often difficult to use. In a dynamic register, the value is set by trapping a charge on a small capacitor. This small capacitor has limited driving capacity, its charge is destroyed when it is read, and it slowly leaks and therefore must be refreshed periodically. These are characteristics are all acceptable for use in our register file design. The limited driving capacity is acceptable because only a small number of cells are on any wire and these wires all short. Destructive read is acceptable because register values are re-written when a thread completes its active phase. And refreshing is not needed because the threads and their registers are constantly being cycled through the processor.

A minor limitation of this design is that there is a minimum period after a context switch before the processor can context switch again. This minimum period is 4 cycles for a 5 stage pipeline like the MIPS R3000 and is shown in Figure 8.5. The grey thread (which uses register bank B) executes for the minimum period of 4 cycles, during which the A register bank is in use every cycle either by the white or black thread or for saving and restoring registers. This 4 cycle minimum on the context switch interval should not pose a problem since context switches rarely occur this frequently.

8.2 Hardware for Switch-On-Miss

This section describes the additions and changes to our multithreaded processor that are needed to support the switch-on-miss multithreading model. A revised processor datapath is shown in Figure 8.6. The main change is the addition of a cache for shared data and support for cache coherency. Since the processors are multithreaded, the caches must be *lock up free* so that they can continue operating while misses are being serviced in the network.

8.2.1 Cache Coherency

Supporting cache coherency is complex[HLRW92], and some machines, such as CRAY's parallel vector computers and the TERA computer, choose to put their complexity

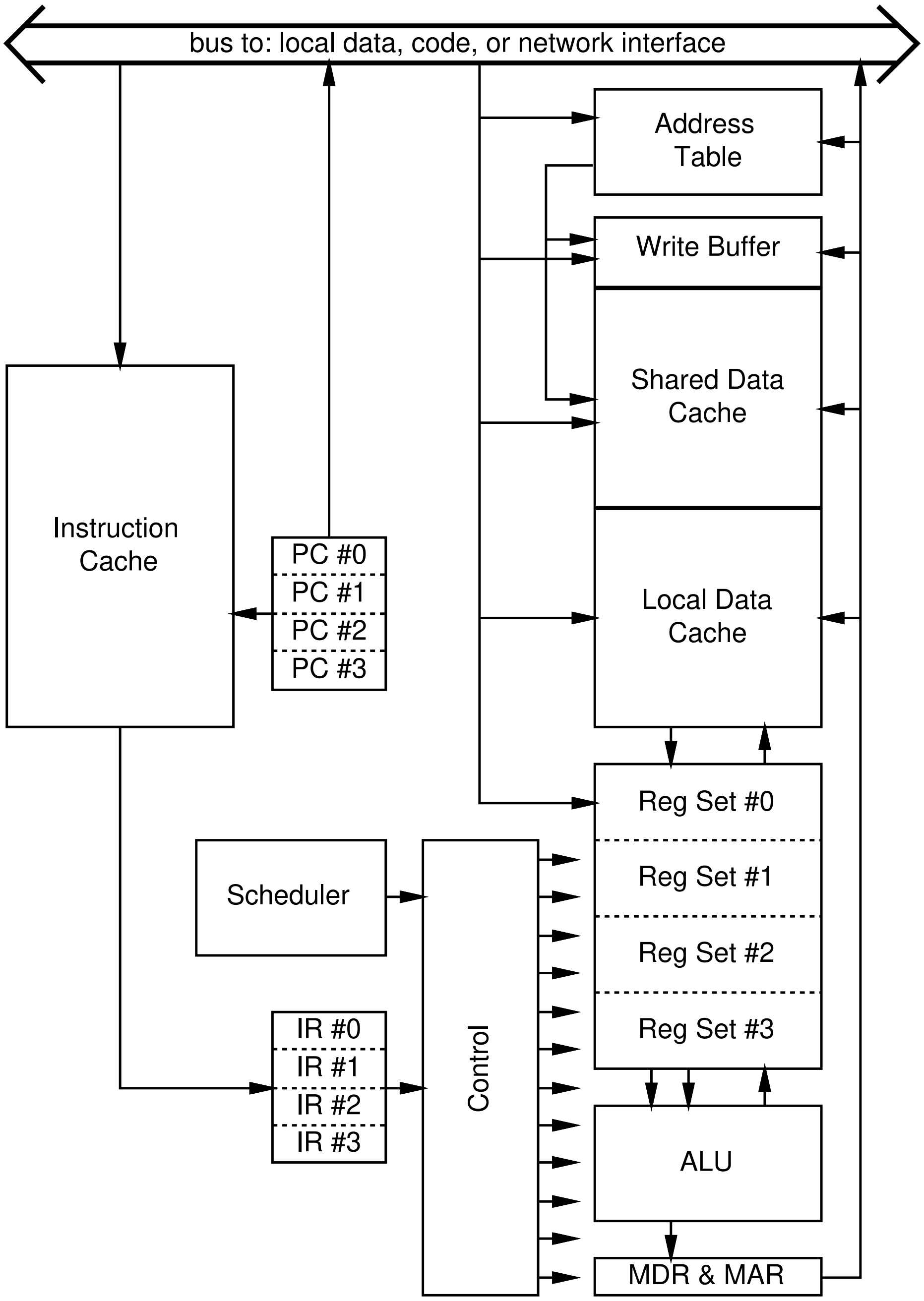

Figure 8.6: Datapath with changes for switch-on-miss multithreading.

elsewhere and build machines without caching of shared data. However, we suggested in Chapter 6 that it may be more cost effective to provide shared data caches and cache coherency than to build networks that provide enough bandwidth to satisfy the needs of large cacheless multiprocessors.

Throughout this dissertation we have used the Censier & Feautrier[CF78] (C&F) invalidation protocol. This is a conceptually simple protocol, and a slight variation was used in the 64 processor DASH prototype[LLJ+92]. However, for a large parallel machine, C&F is impractical to implement in hardware because each memory line must be accompanied by a full directory vector that indicates which caches hold a copy of that line. For a 1024 processor machine, the overhead would be 1024 directory bits for each 16 byte (128 bit) data line.

More recent studies have suggested that variants of the C&F protocol can obtain nearly the same performance with reasonable hardware cost[ASHH88, ON90]. Since the performance is nearly the same, we chose to simulate the conceptually simpler C&F protocol.

8.2.2 Result Matching and the Address Table

In Section 8.1.2 we introduced the idea of sending a small tag along with each remote reference, and upon the return of the result, using that tag to put the result directly into the register file. This direct storage of results into the register file makes the implementation of cache coherency slightly easier by eliminating a possible race condition. This race occurs when a result is placed only in the cache and thus must remain there until the requesting thread has a chance to use it. If some other thread somehow causes the result to be evicted from the cache, the requesting thread will find it missing and have to fetch it again. It is possible that this could result in deadlock[GHG+91]. To avoid deadlock, the results must be locked into the cache until they are used by the requesting thread. Thus the simplification afforded by placing results directly into the register file is that the cache is relieved of its obligation to lock data until it is used.

These tags are also important in conserving network bandwidth. Normally an outgoing read message would contain the address, and the return message would contain both the address and the data. The address in the return message is needed to store the data into the cache. The returning of the address can be eliminated, however, by saving the address in a local table and retrieving it using the message tag upon the result's return.

This table is the address table shown in Figure 8.6. A similar idea was proposed by Dickey & Kenner[DK92] where the tag was called the *Outstanding Request Index*.

8.2.3 Stalling On Writes

He have assumed throughout our simulations that the processor never stalls on writes. For a machine without caching, writes are simply issued into the network and the processor proceeds. Since writes do not return a result, there is usually no reason to stall (or context switch to another thread).

The one case where one might need to stall the processor is to satisfy the requirements of the memory consistency model. For example, to provide sequential consistency[Lam79], all accesses must appear as if they were executed in some global order that is consistent with a sequential interleaving of the thread executions. If the network does not preserve the ordering of accesses, then to be safe a processor must stall after each remote access until that access is completed. Most networks do not preserve ordering of accesses, but some such as the Fluent[Ran89] network do.

Alternatively, the processor can avoid stalling by providing a weaker consistency model such as weak consistency[DSB88] or release consistency[GLL$^+$90]. These are applicable to programs that use explicit synchronization primatives (such as locks) to coordinate the interchange of data among threads. This class of programs (to which all of our applications belong) are called data race free[AH90]. For these programs, waiting for accesses to complete is only required at synchronization points. This effectively eliminates almost all stalls on writes.

An additional cause for stalling occurs on systems with caching. When a processor tries to write a value, the processor must first have exclusive ownership of the value's cache line. If the cache does not contain the line (or exclusive ownership status), it must be obtained (over the network) before the write can be performed. In order to allow the processor to continue executing, the pending write can be put into a write buffer[PH90]. This buffer holds the write operation until the cache line is obtained, and then it performs the write.

For the simulation studies in this dissertation, we have assumed a write buffer of sufficient size to hold all pending writes, and thus we never stall or context switch the processor because of a write to shared memory.

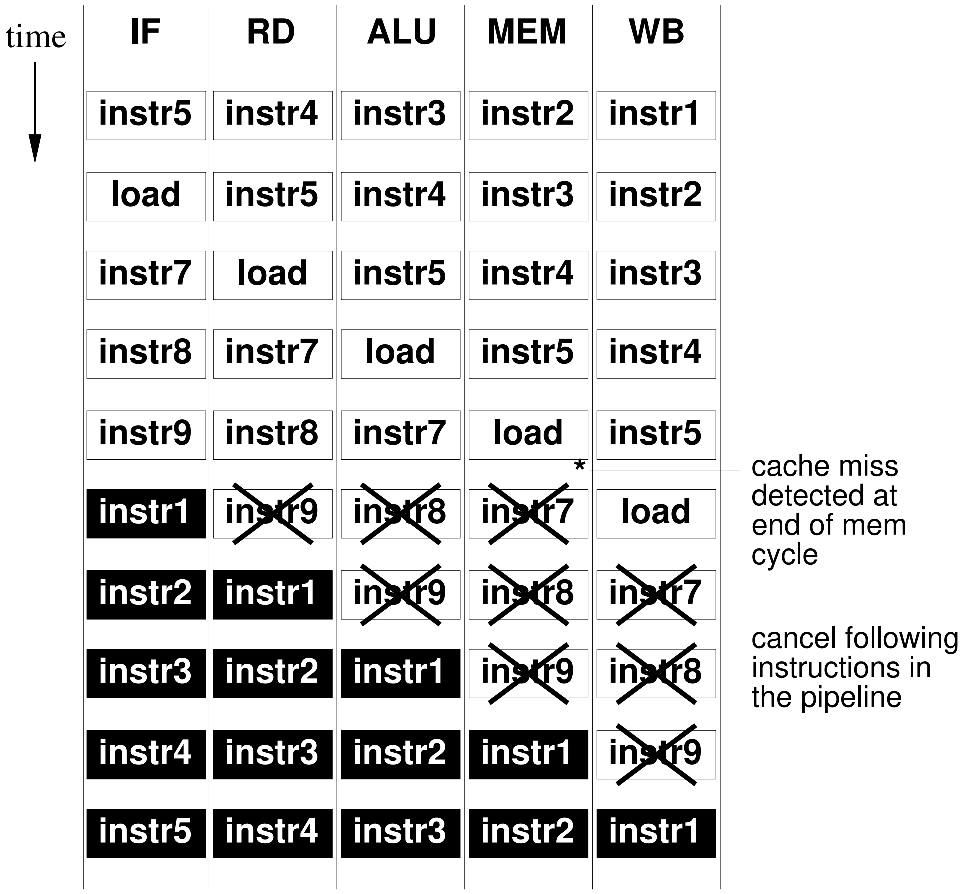

Figure 8.7: Pipelined context switch for switch-on-miss.

8.2.4 Context Switch Delay

For the **switch-on-miss** multithreading model, a context switch can no longer be performed in a single cycle as it was for the **explicit-switch** model as described in Section 8.1.1. The difference is that now the context switch decision is based on the detection of a cache miss, which occurs deep in the processor pipeline, rather than during the decoding of the instruction opcode, which occurs near the start of the pipeline.

Figure 8.7 shows a pipelined context switch that is caused by a cache miss. The sixth instruction of the white thread is a load instruction that cause a miss, and this miss is detected at the end of the MEM stage. At this point three subsequent instruction have already entered the pipeline. These three instruction will be canceled, and thus the cost of the context switch is three cycles.

Application (multithreading)	Efficiency switch cycles: on miss = 3 on timeout = 0	Efficiency switch = 8 cycles	Loss
sieve (mt = 1)	89	86	3%
blkmat (mt = 3)	79	77	2%
sor (mt = 4)	88	85	3%
ugray (mt = 3)	88	85	3%
water (mt = 3)	91	90	1%
locus (mt = 2)	83	78	5%
mp3d (mt = 11)	86	77	11%
barnes (mt = 2)	82	80	2%

Table 8.2: Switch-On-Miss: Performance loss with 8 cycle context switch.

If, however, the context switch is caused by a timeout rather than a cache miss, the context switch can be performed at the start of the pipeline rather than from deep within it. This means that a context switch caused by a timeout can be fully pipelined and thus without any wasted cycles.

Table 8.2 shows the performance loss if all context switches take 8 cycles instead of either 3 or 0 as just explained. The experimental parameter are otherwise the same as in the studies of **switch-on-miss** (refer to Table 5.6. Excluding `mp3d`, which does not cache well, the performance loss due to the longer context switch is typically 2% or 3% with the worst case being 5% for locus. In fact, the performance loss is somewhat overstated here because of the timeout switches. Other scheduling policies that introduce fewer spurious context switches would be able to mitigate the performance loss further.

These results show that a fast pipelined context switch provides only small performance gains over a less aggressive implementation that drains the pipeline before staring the next thread. This smaller performance impact of context switch time, compared to that under explicit-switch, results from the longer run-lengths between context switches.

8.3 Conclusions and Extension to Multiprogramming

In this chapter we have indicated that the hardware mechanisms needed to build an **explicit-switch** or **switch-on-miss** multithreaded processor are reasonable. We have presented them as modifications and additions to a simple RISC processor, and we have suggested that multithreaded processors might be designed by modifying current microprocessor designs. Some researchers have recently proposed that in the future even uniprocessors should be multithreaded[CGL92, FP91, LGH92] because of increasing memory latencies and the increased difficulty of scheduling deeper and wider pipelines.

Some of the simplicity of the multithreading support hardware presented in this chapter comes from the fact the we have only considered the parallel machine to be running a single application at a time. If instead we had tried to provide a more general parallel processor, such as TERA[ACC+90], where each processor could be shared by threads from several different programs, there would have been additional hardware complexities. TERA allows up to 128 threads per processor and 16 simultaneously executing programs. The large number of threads requires a very large register file. And the simultaneous execution of multiple programs requires memory protection to protect all the threads from each other.

Multiprogramming is a very desirable attribute of a parallel machine. At times the machine must clearly be devoting 100% of its resources to solving a single large problem, for otherwise such a large machine would not be needed. But often the machine must support the simultaneous development and testing of applications by many programmers. We believe the approach taken by the CM-5 is a good compromise. The CM-5 allows the machine to be partitioned into smaller machines for independent use, and it also allows the entire machine to be time sliced between applications. This time slicing can be done at intervals in the range of seconds, to limit the throughput loss due to stopping the machine, draining the network, and switching to a new process.

Partitioning the machine allows a real time user to collar a portion of the machine, whereas time slicing allows multiple application developers to share the machine while testing their applications at the full machine size. Under these policies, the individual processors are always executing just a single program at a time, and thus the processor model presented in this chapter is adequate for building real parallel machines.

An additional benefit of multithreading over current parallel machines is that the number of processors assigned to an executing program can be easily changed. For example, consider a program that is being run at a multithreading level of $M = 2$ and is using all 1024 processors. If a second program is started, the first program can be compressed down to use 512 processors at $M = 4$, or 256 processors at $M = 8$. Without multithreading it is difficult to change the number of processors since somehow the running program must be reconfigured to use fewer threads.

Chapter 9

Conclusions and Future Directions

In this dissertation we have identified long memory latency and limited network bandwidth as the main problems impeding the development of large shared memory multiprocessors. It appears inevitable that any large interconnection network will have long latencies because of the many nodes, links, and delays which messages will encounter on their transit through the network. Also any network that provides some level of scalable bandwidth will represent a large fraction of the hardware in a large system, and thus cost effectiveness will dictate that bandwidth be limited.

We do not expect that any technological innovation in network design will be able to eliminate the latency and bandwidth problems, so instead we have focused on making the processor tolerant of long latencies and reducing and quantifying the amount of network bandwidth needed.

9.1 Conclusions

The main focus of this research has been on models of multithreaded processors whose design is similar to current state-of-the-art microprocessors except for the addition of support for fast context switching. These multithreaded processors will execute instructions in the same pipelined (and superscalar) fashion as current microprocessors except that at certain instructions (determined by the particular multithreading model) they will quickly context switch to another thread rather than wait for completion of a remote reference.

For machines without caching of shared data, we have looked at the **switch-on-load** and **explicit-switch** multithreading models. **Switch-on-load** performs poorly because context switches occur frequently and often very close together. This means that a large number of threads will be needed to hide the latency, and because of the many short run-lengths, sometimes the latency still will not be completely hidden.

Explicit-switch improves performance by introducing an explicit context switch instruction. This instruction can be used by an optimizing compiler to group together independent shared memory references. This grouping allows a single thread to issue multiple references into the network before switching to another thread. The run-lengths are increased, and the distributions are significantly improved by the elimination of most short run-lengths.

Our results show that **explicit-switch** is able to tolerate latencies of 200 cycles by using a multithreading level of 10 threads (or less) per processor, and that this is sufficient to allow all of the applications studied to obtain efficiencies of 80%.

However, since there is no caching of shared memory in these systems, all shared references are sent across the network, and thus the resulting network bandwidth demands can be quite high. These bandwidth demands vary considerably across the applications and range as high as 30 bits/operation. We expect that providing such high bandwidth will be expensive, and thus conclude that although multithreaded systems without shared data caching can achieve high execution efficiencies, they may not be cost effective because of their high bandwidth requirements.

Caching was effective for all but one of our applications (which has since been rewritten to improve its caching behavior). For the rest of the applications, caching was able to reduce the average bandwidth requirement to under 2 bits/operation. This large reduction in bandwidth suggests that the cost and complexity of maintaining coherent caches on a large machine will be justified by the savings afforded from use of a skinnier network.

Caching is also beneficial in that it filters out many of the remote references and thus eliminates many potential long latency operations. Typical miss rates ranged from 1% to 4%. With so many fewer remote references, the impact of long latency is diminished. Our results show that with latencies of 200 cycles, execution efficiencies of 60% to 70% can be achieved without multithreading, and that with a multithreading level of 3 threads per processor, efficiencies can be raised to 80% to 90%.

Multithreading systems both with and without caching are able to achieve efficiencies of 80%. The advantage of caching is that it reduces the multithreading level and the amount of network bandwidth that is required.

While most of our experiments have assumed that there will be adequate network bandwidth available, we also looked at the impact on performance of having limited bandwidth, as will be the case for real networks. Our results for large (256 processor) systems (using both multithreading and caching) show that network traffic will be bursty. If the system is to have at most minor performance degradation, then the network will need to supply a remote reference bandwidth of 2 to 4 bits/operation and a memory module bandwidth of 8 to 16 bits/operation. The higher memory module bandwidth is necessary because of random hot spot congestion.

9.2 Future Directions

We expect in the future that the latency problem will continue to increase. Processors will continue to get faster, and because of both higher clock rates and superscalaring, they will issue remote references at higher rates. Ever larger parallel machines will also be desired and these will have larger networks and longer latencies. These faster processors and larger networks will both contribute to an increased need for latency tolerance.

Without caching, our simulations show that tolerating a 200 cycle latency requires 10 threads per processor. We expect that the number of threads needed will grow at least linearly with latency, and thus at significantly larger latencies, the number of threads needed may grow prohibitively large. An important aspect of these machines that needs further research is the amount of inter-block grouping that can be obtained by a smart compiler. Our estimates in Section 4.3.2 suggests that research in this area should be successful.

With caching, our simulations in Section 7.4 show that with longer latencies the performance of a single threaded processor drops to 30% efficiency at a latency of 1,000 cycles. At this point multithreading is very beneficial and can double the performance to 60% efficiency, while still using just a moderate number of threads per processor.

In the introduction of this dissertation we listed six mechanisms for reducing the impact of memory latency: caching, multithreading, weak consistency, prefetching, layout, and aggregation. In this dissertation have focused primarily on the first three techniques. These are the most hardware oriented and the most broadly applicable, but there is potential benefit from all of these mechanisms, and we suggest future research should look at exploiting all of these techniques.

Bibliography

[ACC$^+$90] Robert Alverson, David Callahan, Daniel Cummings, et al. The TERA Computer System. In *1990 Int. Conf. on Supercomputing*, pages 1–6, 1990.

[AG89] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings Publishing Co., 1989.

[Aga92] Anant Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, pages 525–539, September 1992.

[AH90] Sarita V. Adve and Mark D. Hill. Weak Ordering — A New Definition. In *The 17th Annual Int. Symp. on Computer Architecture*, pages 2–14, 1990.

[ALKK90] 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.

[And90] Thomas E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, pages 6–16, January 1990.

[ASHH88] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *The 15th Annual Int. Symp. on Computer Architecture*, pages 280–289, 1988.

[BBN89] BBN Advanced Computers Inc. *TC2000 Technical Product Summary*, 1989.

[BDCW91] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, September 1991.

[Bel85] C. Gordon Bell. Multis: A New Class of Multiprocessor Computers. *Science*, 228:462–467, April 26 1985.

[BJS88] F. Baskett, T. Jermoluk, and D. Solomon. The 4D-MP Graphics Superworkstation: Computing + Graphics = 40 MIPS + 40 MFLOPS and 100,000 Lighted Polygons per Second. In *Proc. Compcon Spring 88*, pages 468–471, 1988.

[Boo89] Bob Boothe. Multiprocessor Strategies for Ray-Tracing. Master's thesis, U.C. Berkeley, September 1989. Report No. UCB/CSD 89/534.

[BR90] Roberto Bisiani and Mosur Ravishankar. PLUS: A Distributed Shared-Memory System. In *The 17th Annual Int. Symp. on Computer Architecture*, pages 115–124, 1990.

[BR92] Bob Boothe and Abhiram Ranade. Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessors. In *The 19th Annual Int. Symp. on Computer Architecture*, pages 214–223, May 1992.

[C+88] R. C. Covington et al. The Rice Parallel Processing Testbed. In *Proc. 1988 ACM SIGMETRICS*, pages 4–11, 1988.

[CF78] 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.

[CGL92] David E. Culler, Michial Gunter, and James C. Lee. Analysis of Multithreaded Microprocessors under Multiprogramming. Technical Report 92/687, Computer Science Division, University of California, Berkeley, 1992.

[CSS+91] 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.

[Dal90] William J. Dally. Virtual-Channel Flow Control. In *The 17th Annual Int. Symp. on Computer Architecture*, pages 60–68, 1990.

[Del91] Chrysanthos N. Dellarocas. A High-Performance Retargetable Simulator for Parallel Architectures. Technical Report MIT/LCS/TR-505, Massachusetts Institute of Technology, June 1991.

[DGH91] Helen Davis, Stephan R. Goldschmidt, and John Hennessy. Multiprocessor Simulation and Tracing using Tango. In *Proc. 1991 Int. Conf. on Parallel Processing*, pages II 99–107, 1991.

[DK92] Susan R. Dickey and Richard Kenner. Hardware Combining and Scalability. In *Proc. of the 1992 ACM Symposium on Parallel Algorithms and Architectures*, pages 296–305, 1992.

[DM93] Jack Dongarra and Hans Meuer. Top500 computers - a new competition. posted to comp.parallel, May 1993.

[DRPS87] F. Darema-Rogers, G. F. Pfister, and K. So. Memory Access Patterns of Parallel Scientific Programs. In *Proc. 1987 ACM SIGMETRICS*, pages 46–58, 1987.

[DSB88] Michel Dubois, Christoph Scheurich, and Favé A. Briggs. Synchronization, Coherence, and Event Ordering in Multiprocessors. *Computer*, pages 9–21, February 1988.

[Enc87] Encore Computer Corporation. *Multimax Technical Summary*, 1987.

[FP91] Matthew K. Farrens and Andrew R. Pleszkun. Strategies for Achieving Improved Processor Throughput. In *The 18th Annual Int. Symp. on Computer Architecture*, pages 362–369, 1991.

[Fuj88] Fujitsu. *SPARC MB86901 (S-25) High Performance 32-Bit RISC Processor*, September 1988.

[GGK+82] 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.

[GHG+91] 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.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Lauden, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *The 17th Annual Int. Symp. on Computer Architecture*, pages 15–26, 1990.

[GT90] Gary Graunke and Shreekant Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *Computer*, pages 60–69, June 1990.

[Gus88] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, pages 532–533, May 1988. Technical Note.

[Gus92] David B. Gustavson. The Scalable Coherent Interface and Related Standars Projects. *IEEE Micro*, pages 10–22, February 1992.

[GVW89] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *ASPLOS-III Proceedings*, pages 64–75, 1989.

[HF88] 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.

[Hig93] High Performance Fortran Forum. High Performance Fortran Language Specification. Technical Report CRPC-TR 92225, Center for Research on Parallel Computation at Rice University, May 1993.

[HLRW92] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. In *ASPLOS-V Proceedings*, pages 262–273, 1992.

[HLS92] Maurice Herlihy, Beng-Hong Lim, and Nir Shavit. Low Contention Load Balancing on Large-Scale Multiprocessors. In *Proc. of the 1992 ACM Symposium on Parallel Algorithms and Architectures*, pages 219–227, 1992.

[Kan89] Gerry Kane. *MIPS RISC Architecture.* Prentice Hall, 1989.

[Ken92] Kendall Square Research Corporation. *Kendall Square Research Technical Summary*, 1992.

[Kow85] Janusz S. Kowalik, editor. *Parallel MIMD computation : the HEP supercomputer and its applications.* MIT Press, 1985.

[KS93] R. E. Kessler and J. L. Schwarzmeir. CRAY T3D: A new Dimension for Cray Research. In *COMPCON SPRING '93*, pages 176–181, 1993.

[LAD+92] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, et al. The Network Architecture of the Connection Machine CM-5. In *Proc. of the 1992 ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, 1992.

[Lam79] Leslie Lamport. How to Make a Multiproccessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, pages 690–691, September 1979.

[Lar90] James R. Larus. SPIM S20: A MIPS R2000 Simulator. Technical report, C. S. Dept., University of Wisconsin-Madison, 1990.

[Lei85] C. E. Leiserson. Fat-Trees: Universal Network for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, pages 892–901, October 1985.

[LGH92] James Laudon, Anoop Gupta, and Mark Horowitz. Architectural and Implementation Tradeoffs in the Design of Multiple-Context Processors. Technical report, Stanford University, 1992. Technical Report CSL-TR-92-523.

[LLG+90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Cache Coherence Protocal for the DASH Multiprocessor. In *The 17th Annual Int. Symp. on Computer Architecture*, pages 148–159, 1990.

[LLG+92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolk-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash Multiprocessor. *Computer*, pages 63–79, March 1992.

[LLJ+92] Daniel Lenoski, James Lauden, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Implementation and Performance. In *The 19th Annual Int. Symp. on Computer Architecture*, pages 92–103, 1992.

[LM92] Thomas J. LeBlanc and Evangelos P. Markatis. Shared Memory vs. Message Passing in Shared-Memory Multiprocessors. In *Fourth IEEE Symp. on Parallel and Distributed Processing*, 1992. to appear in SPDP 92.

[LS91] Steven Lucco and Oliver Sharp. Parallel programming with coordination structures. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 197–208, Orlando, Florida, January 1991.

[LT88] Tom Lovett and Shreekant Thakkar. The Symmetry Multiprocessor System. In *Proc. 1988 Int. Conf. on Parallel Processing Vol. 1 Architecture*, pages 303–310, 1988.

[LYL87] Roland L. Lee, Pen-Chung Yew, and Duncan H. Lawrie. Data Prefetching in Shared Memory Multiprocessors. In *Proc. 1987 Int. Conf. on Parallel Processing*, pages 28–31, 1987.

[Man92] Daniel Mann. UNix and the Am29000 Microprocessor. *IEEE Micro*, pages 23–31, February 1992.

[Mar87] Don Marsh. UgRay - An Efficient Ray-Tracing Renderer for UniGrafix. Master's thesis, U.C. Berkeley, May 1987. Report No. UCB/CSD 87/360.

[MCS91] John M Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, pages 21–65, February 1991.

[MIP86] MIPS Computer Systems. *MIPS language programmer's guide*, 1986.

[MIP91] MIPS Computer Systems, Inc. *MIPS R4000 Microprocessor User's Manual*, 1991.

[ML92] Evangelos P. Markatos and Thomas J. LeBlanc. Shared-Memory Multiprocessor Trends and the Implications for Parallel Program Performance. Technical Report 420, University of Rochester, May 1992.

[MPS87] Theodore E. Mankovitch, Val Popescu, and Herbert Sullivan. CHoPP Principles of Operation. In *Proc. Supercomputing '87 Vol. I*, pages 2–10, 1987.

[NL91] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(8):52–60, August 1991.

[NPA92] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *The 19th Annual Int. Symp. on Computer Architecture*, pages 156–167, 1992.

[O'K89] Brian W. O'Krafka. An Empirical Study of Three Hardware Cache Consistency Schemes for Large Shared Memory Multiprocessors. Technical report, Electronics Research Laboratory, University of California, Berkeley, May 1989. Tech Report UCB/ERL M89/62.

[O'K92] Brian Walter O'Krafka. *Design and Evaluation of Directory-Based Cache Coherence Systems.* PhD thesis, U. C. Berkeley, January 1992.

[ON90] 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.

[Ost89] Anita Osterhaug, editor. *Guide to Parallel Programming on Sequent Computer Systems.* Prentice Hall, 1989.

[PC90] 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.

[PGH+84] Steven A. Przybylski, Thomas R. Gross, John L. Hennessy, Norman P. Jouppi, and Christopher Rowen. Organization and VLSI Implementation of MIPS. Technical Report 84-259, Computer Systems Laboratory, Stanford, 1984.

[PH90] David A. Patterson and John L. Hennessy. *Computer Archtecture a Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[PN85] G. F. Pfister and V. A. Norton. "Hot Spot" Contention and Combining in Multistage Interconnection Networks. In *Proc. 1985 Int. Parallel Processing Conf*, pages 790–797, 1985.

[Ran89] Abhiram Gorakhanath Ranade. *Fluent Parallel Computation.* PhD thesis, Yale University, May 1989.

[RL92] Anne Rogers and Kai Li. Software Support for Speculative Loads. In *ASPLOS-V Proceedings*, pages 38–50, 1992.

[RR93] M. T. Raghunath and Abhiram Ranade. Customizing Interconnection Networks to Suit Packaging Hierarchies. Technical Report UCB/CSD–93–725, Computer Science Division, University of California, Berkeley, CA 94720, January 1993.

[RT86] R. Rettberg and R. Thomas. Contention is no obstacle to shared-memory multiprocessing. *Communications of the ACM*, 29(12):1202–1212, Dec 1986.

[SBC91] Rafael H. Saavedra-Barrera and David E. Culler. An Analytic Solution for a Markov Chain Modeling Multithreaded Execution. Technical Report 91/623, Computer Science Division, University of California, Berkeley, 1991.

[SBCvE90] 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.

[SHG92] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Implications of Hierarchical N-Body Techniques for Multiprocessor Architecture. Technical Report CSL-TR-92-506, Stanford University, 1992.

[Smi82] A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[Smi92] Burton Smith, 1992. personal communication.

[SWG92] Jaswinder Pal Singh, Wolk-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, pages 5–44, March 1992.

[TD91] Manu Thapar and Bruce Delagi. Cache Coherence for Large Scale Shared Memory Multiprocessors. *Computer Architecture News*, pages 114–119, March 1991.

[TKB92] Andrew S. Tanenbaum, M. Frans Kaashoek, and Henri E. Bal. Parallel Programming Using Shared Objects and Broadcasting. *Computer*, pages 10–19, August 1992.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *The 19th Annual Int. Symp. on Computer Architecture*, pages 256–266, 1992.

[Waw91] John Wawrzynek, 1991. personal communication.

[WG89] 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*, pages 273–280, 1989.

[Wol89] Ronald W. Wolff. *Stochastic Modeling and the Theory of Queues*. Prentice Hall Inc., 1989.

[YTL86] 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.

Appendix A

Distribution Function Histograms

Normally, distribution functions are shown either as cumulative distribution functions or as histograms, where both axis have linear scales. These turned out to be inadequate for showing the properties of the distributions which arose in this research, so as an alternative, we have developed a new type of histogram that represents data primarily by area rather height.

A.1 Area Proportional to Value

Figure A.1 shows an example of our new histogram with various data points ranging from 50% to 0.05% of the total data.[1] The data points are represented as two-dimensional triangular piles whose area is proportional to their value. You can think of

[1] Please excuse the fact that the piles do not sum to exactly 100% in this example.

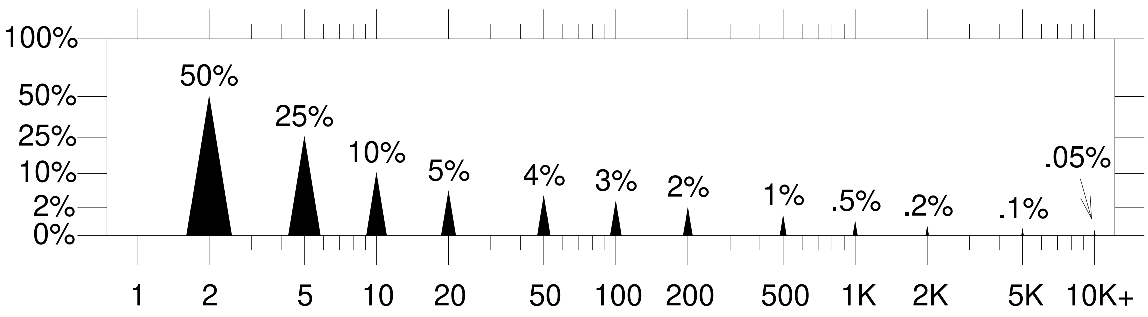

Figure A.1: Example histogram showing piles of various sizes.

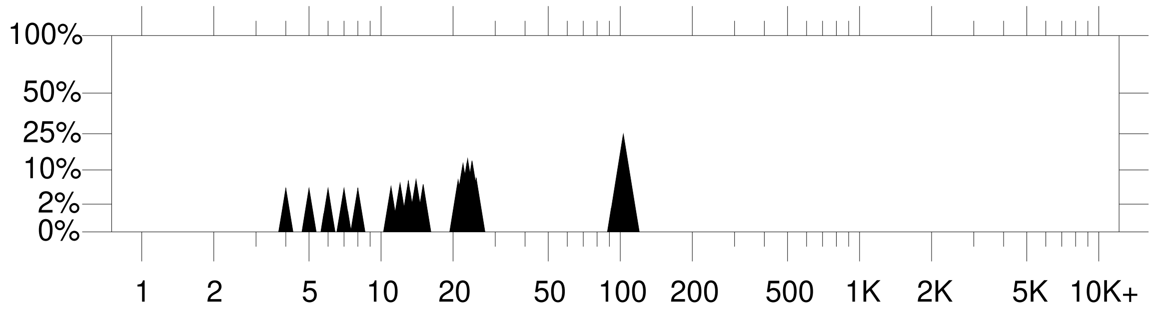

Figure A.2: Example histogram showing aggregation of adjoining piles.

these as piles of sand, where the proportion of sand corresponding to each data point was dropped from above that point. The triangular shape of the piles comes from the critical slope of the sand. The height of a pile will be equal to the square root of its area (value), and thus a square root scale is marked on the vertical axis. The advantage of representing data by area rather than height is that values ranging over three orders of magnitude can all be distinguished from each other, yet there is no distortion as there would have been if we had tried to use a logarithmic vertical axis. For some distributions that come up, it is important to see contributions of less than one percent because if they occur at large values, they can have a large impact on the mean.

A.2 Aggregation of Adjoining Piles

Another important property of these histograms is the ability to combine nearby piles together when appropriate. For example, if 1% of the run-lengths were spread out evenly over the range from 2000 to 2100, each point in this range would have a value of less than 0.01%, and the individual points would be indistinguishably small. Together, however, they constitute a sizable amount and should definitely appear as such. Since the points from 2000 to 2100 are all so close together on the logarithmic scale, it is appropriate to draw just a single pile constituting 1% which is centered over the range 2000 to 2100.

Using the sand analogy, if several piles are close enough together that they would overlap, the sand should simply pile up higher and the piles should start to merge together. Figure A.2 shows an example of this merging. There are four clusters of five piles each, and each pile constitutes 5% of the data. The first cluster is the points {4,5,6,7,8}, and these

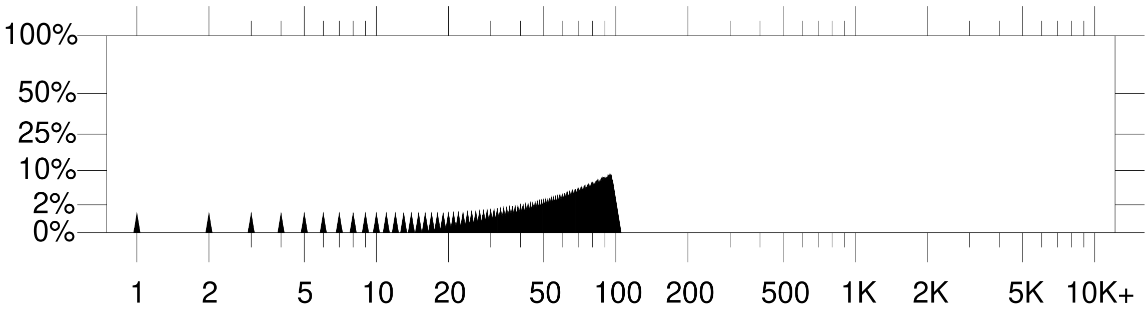

Figure A.3: Example histogram showing a uniform distribution over the domain [1,100].

points are far enough apart on the logarithmic scale that the piles remain independent. The second cluster of points is {11,12,13,14,15}. These piles are starting to merge, but are still distinguishable. The third cluster of points is {21,22,23,24,25}, and these are close enough together that they have almost merged into a single pile. Finally, the fourth cluster of points at {101,102,103,104,105} are so close together that the merged pile looks identical to a single pile of 25%. The important point is that each of the four piles comprises a total of 25% and uses the same amount of ink on the page. Where there is room to distinguish the individual components, this is done, and when there is not enough room, the pile are aggregated into a larger pile.

A.3 Mismatch with Old Intuition

Unfortunately, if we are familiar with the look of a distribution when plotted on linear axis, it will appear distorted when plotted in this new format. A strong example of this is shown in Figure A.3. Here we show a uniform distribution where each point from 1 to 100 has 1% of the run-lengths. On linear axis this would appear flat. But in our new format, the tighter spacing at the high end does not allow enough room to show the many small piles independently. These adjoining piles are aggregated together and the graph rises.

Despite this mismatch with our old intuition, in most cases these graphs provide a compact and clear understanding of the distributions that we will see in this dissertation.

Appendix B

Simulator

B.1 Introduction

Simulation is an essential tool in the process of computer design. While the speed of simulation has always been a concern, it is of critical concern when simulating parallel machines because of the increased computational power of these machines. The arithmetic is obvious: simulating one second of execution of a one MIP uni-processor requires simulating one million instructions, but simulating one second of execution of a thousand processor parallel machine requires simulating one billion instructions. Most simulation based research reports being limited in scope and accuracy by the speed of their simulators[BR92, GHG+91, ON90]. Faster simulators allow larger and more realistic simulations to be performed and help speed up the experimental process by allowing more rapid feedback of simulation results.

Our simulation system, **FAST** (Fast Accurate Simulation Tool), has a simulation slowdown ranging from 10 to 100. This *slowdown* factor is the average number of cycles it takes to simulate a single cycle of execution for a single processor. It varies based on the application program being simulated. Applications with more frequent references to shared memory interact with the simulator more frequently and therefore take longer to simulate. Comparable simulation systems such as that of O'Krafka[O'K89] or Tango[DGH91] have reported slowdowns of 2,000 and 500-6,000 respectively.

FAST was developed for the purpose of studying large shared memory multiprocessors with hundreds or thousands of processors, and to run real applications on these simulated machines. To support our simulation studies of such large systems, we needed a simulator that was orders of magnitude faster than the other simulators that were available at the time of its development.

The technique of *execution driven simulation*[C+88] is the foundation of FAST. We are not concerned with the simulation of a new instruction set, but rather we are concerned with higher level aspects of the simulated machine. Because of this, we can accept the instruction set of the host machine on which we are performing our simulations. This allows us to directly execute most instructions instead of spending hundreds of cycles to interpret each instruction individually[Lar90]. The assembly code of the application program is augmented with additional instructions which keep track of a thread's execution time and return control to the simulator at special events such as references to shared memory. The net result is that most instructions are directly executed in a single cycle, and only the small fraction of instructions which interact with the rest of the system need to be simulated.

In this research we have extended the idea of execution driven simulation with several new techniques that have allowed up to build a simulator that is both faster and more accurate than previous comparable simulators.

B.1.1 Overview

The remainder of this appendix is broken into five sections. Section B.2 discusses several previous simulators and their tradeoffs in performance, accuracy, and other concerns. Section B.3 presents an overview of FAST. Section B.4 explains our extensions to the ideas of execution driven simulation. Section B.5 reports performance results. And section B.6 summarizes and suggests directions for future research.

B.2 Previous Simulators and Tradeoffs

There have been an enormous number of simulation systems written for various purposes. Here we focus on a few recent simulators that have all been designed for basically the same purpose: simulating large shared memory multiprocessors at the instruction level.

We compare their performance in terms of their slowdown factors, and we also look at two aspects of accuracy. One is the degree to which instruction timings reflect that of an actual processor, and the second is the degree to which shared memory references are interleaved and simulated in an accurate global order.

B.2.1 Cycle-by-Cycle Simulators

The most straight forward type of simulator to build is one that cycles through the parallel processors, simulating one instruction at a time from each of the processors. Two examples are the simulator by O'Krafka[O'K89], which we are more familiar with since this was done at Berkeley, and ASIM(refer to the description in [Del91]) developed at MIT as part of the Alewife project. These simulators are slow because they are essentially assembly language interpreters. The reported slowdown factor for O'Krafka's simulator is 2,000, and for ASIM it is reported as ranging from 200–5,000. Cycle-by-cycle simulators are accurate in interleaving global events since they simulate the entire machine one cycle at a time, but they may be inaccurate in instruction timing (as is O'Krafka's simulator) because it is complex and time consuming to accurately model the processor's pipeline.

The performance of these cycle-by-cycle simulators is dominated by instruction interpretation since this is done for every single cycle of the executed program. Interesting events, like shared memory references, occur less frequently.

B.2.2 Execution Driven Simulators

Execution driven simulation can be substantially faster than a cycle-by-cycle simulator because it eliminates the instruction interpretation portion of the simulator. Instead, control is handed over to the augmented program which executes for several cycles before encountering an event of interest and returning control to the simulator. The simulated processor has now advanced its private clock past those of other simulated processors. Accurate event interleaving dictates that the event should not be processed immediately, but rather it must be scheduled and executed once the entire global state has advanced to the event's time step. This means that instead of cycling between the simulated processors on a cycle by cycle basis, it is sufficient to cycle between them at each event (as long as the events are then queued and later executed at their proper times).

The Tango simulator[DGH91] developed at Stanford is an execution driven simulator. It is based on Unix shared memory and uses Unix context switches in order to switch from executing one processor to another. These heavy weight context switches however require thousands of cycles, and thus they slow the simulator substantially if it switches at every event in order to accurately interleave them. For accurate simulations they report slowdown factors ranging from 500 to 6000. Because of this large cost of context switching, they provide an option to tradeoff accuracy for faster execution by letting the individual processor clocks get out of sync and not trying to accurately interleave the shared memory references. They have recently rewritten their simulator to use a light weight thread package, which should significantly reduce the magnitude of their context switch overhead problem.

The Proteus simulator developed at MIT[BDCW91, Del91] is another execution driven simulator. It does use a light weight thread package, and is substantially faster than Tango. They report typical slowdown factors ranging from 35 to 100. However they have a substantial accuracy problem in their instruction timing because they do not apply code augmentation at a consistent low level. They replace shared memory references in the C source code with calls to the simulation routines (and optionally also insert statistics gathering calls.) They then compile this modified code and apply code augmentation for timing on the assembly language. Because each shared reference (which should be just a single instruction) is replaced with a procedure call, the compiler optimizations that can be applied and the object code produced are substantially changed from that which would have been produced if the original code were compiled directly. In fact, their good performance is partially due to the fact that their insertion of procedure calls causes the compiler to save away important registers, and thus allows them to "exploit 'partial' context switches" in which they only save a limited amount of the register file. This is good for performance, but bad for timing accuracy.

B.2.3 Tradeoffs

We have identified the following five tradeoffs in simulator design:

Performance: Execution driven simulation is the most important factor in building a fast simulator because otherwise the interpretation of individual instructions is the dominant cost. The next most important factor is fast context switching between the simulated processors because frequent context switching is required to accurately order global events.

Accuracy: Performing all code augmentation at the assembly language level is necessary for accurate instruction timing. Any source code modifications that change the code generated by the compiler affect the compiler's optimization ability and the thus accuracy of instruction level timing. Switching between simulated processors at all globals events is required in order to obtain a correct global ordering. If context switching is expensive, then the simulator writer or user is tempted to trade accuracy for performance by context switching less often.

Source Alteration: Ideally the source code should be compiled and optimized in its original form as it would be written for a shared memory multiprocessor. However, all of these simulators require some source changes. Proteus is the most egregious and requires new operators be used for all shared memory references. O'Krafka's simulator and Tango both disallow static shared variables, and thus all such variable must be allocated dynamically and referenced indirectly through pointers. FAST only requires minor syntactic changes[1] that have no affect on the instructions generated by the compiler.

Modularity: All of the simulators have similar modularity. Each allows selecting and mixing different modules for different aspects of the machine: such as the cache and the interconnection network. Normally this is done by linking the modules together, but Tango also has the option (at substantial performance cost) of using distinct Unix processes.

Portability: Portability is poor for all of these systems because they are tied to the instruction set that they are designed for. Direct execution simulators must be run on that specific type of machine, but cycle-by-cycle simulators, since they are interpreters, can use cross-compiled applications and be run on any machine. Porting execution driven simulators to a new machine involves changing the code augmentation to understand the new machine's instruction set. The actual simulators are all written in high level languages and should presumably be portable.

Based on an understanding of these tradeoffs, we have built our FAST simulation system so that it is faster, more accurate, and uses less mutative source alterations. It has similar modularity and portability as in the other simulators discussed.

B.3 Simulator

[1]Most of our applications were originally written for the Sequent[Ost89]. Their syntax for declaring a shared variable is: `shared int x;`. Our syntax is: `int shared_x;`. All uses of the variable `x` are also changed to to `shared_x`. These changes allow shared variables in the object file to be identified by using symbol table information that is normally used for linking.

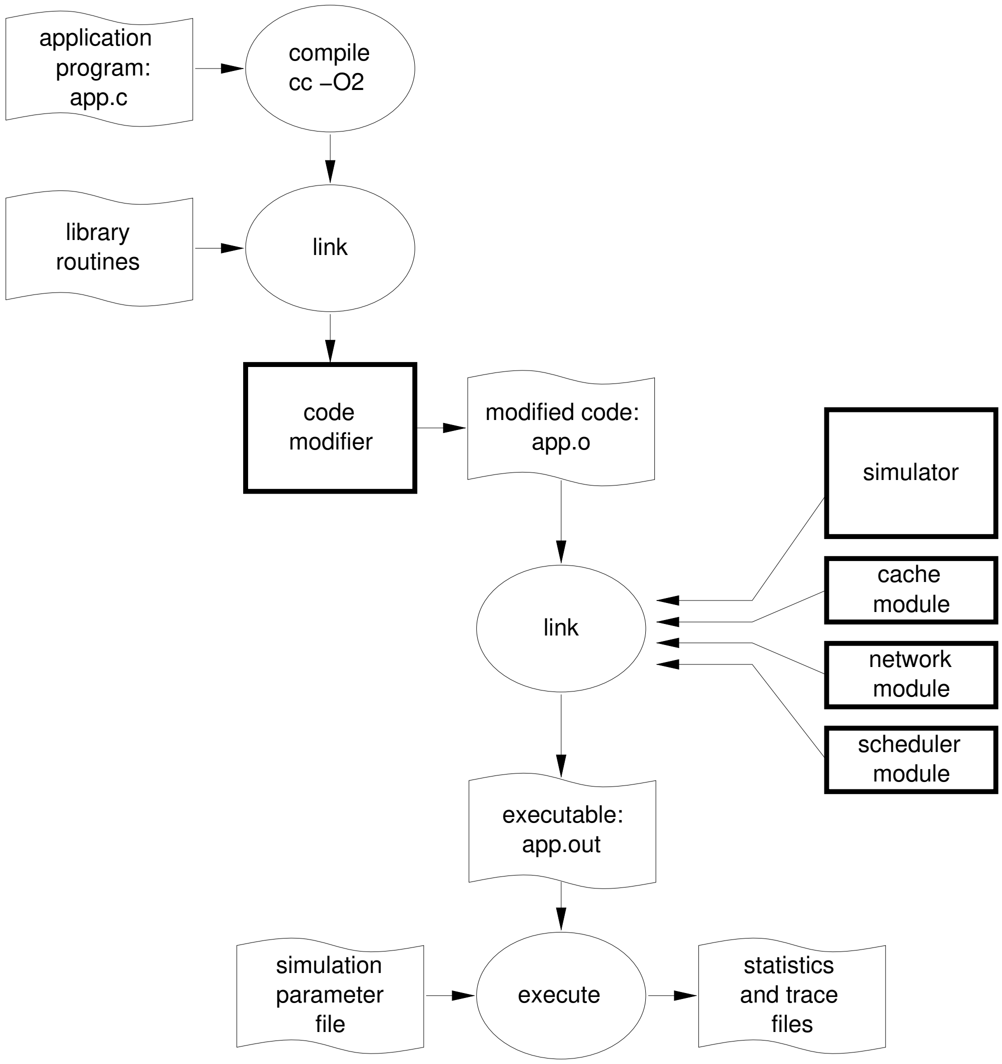

Figure B.1: Diagram of using FAST.

Figure B.1 shows a diagram of using FAST. First the application program to be simulated is compiled with full optimization just as it would be for a real parallel processor, and then it is linked with any libraries that it uses, such as math routines.

The linked object code module is then read into the code modifier which performs the various code augmentations (which will be discussed in the next section). It is important that augmentation be done on library functions since some applications use these extensively. System calls are not handled, but these usually do not occur in the parallel computation phases of the parallel applications that we have studied.

The modified code is then linked with the simulator and selected modules that simulate the caches, network, and scheduler. A large number of these modules have been written, and they can be selected based on what is of interest to the user. For caching there are modules for various cache configurations and protocols, or for no caching at all. For networks the simulator is usually used with a simple constant time network approximation, but it has also been used with a detailed simulator of packet switched networks. The scheduler module is used for multithreading studies and implements simple scheduling policies such as FIFO, or more complex policies like priority scheduling or timeouts.

The single executable file produced includes the simulator, the various modules, and the modified application code. When it is run, the simulator starts first. It reads in a simulation parameter file that specifies the number of processors, level of multithreading, network latency, and other parameters. It then calls initialization routines for the various modules, and then starts up and manages the execution driven simulation of the application program.

The core of the simulator is a simple time wheel scheduler. This is just a linear array with one slot per time step (modulo the array size), where each slot points to a linked list of events that will occur at that time step. The simulator operates by removing an event at the current time step, simulating it (using execution driven simulation), and then placing the resulting event into the proper slot to be executed in the future. This is very efficient since there is no polling to test for ready events. For simulations of large parallel machines, there are so many events that typically every slot has one or more events in it. The average cost of scheduling an event is thus very small.

B.4 Code Augmentation

Code augmentation is the process of taking an original piece of code and adding to it and/or modifying it so that it can perform additional functions. Traditionally it has been used for the following three purposes:

Time Counting: Instructions are added to the each basic block so that when that block is executed, the extra instructions increment a time counter with an amount corresponding to the number of cycles required for the processor to execute the original basic block. This is the basic code augmentation that is used in all execution driven simulators.

Statistics Gathering: Instructions are added to gather statistics such as counts of the number of times that certain pieces of code are executed. This is the basis of execution driven profilers, such as the MIPS `pixie` program[MIP86].

Event Call-Outs: At special events, such as shared memory references, code is inserted to call out to the simulator in order to let the simulator regain control and process the event. This is used in a simplified form when debuggers create breakpoints by replacing the instruction at the breakpoint with a trap instruction.

In this section we extend the idea of augmentation with several new uses:

In-line Context Switching: The augmented code typically runs for just a small number cycles before reaching an event and returning control to the simulator. During this execution only a small subset of the register file is ever accessed, and therefore it is wasteful to actually load and store the entire register set. We use code augmentation to load and store register values at basic block boundaries so that only the used and modified registers are loaded and stored.

Reference Indirection: For a single threaded program, which the compiler thinks it is compiling, static local variables are assigned to fixed memory addresses. However, for a parallel program, each thread needs its own copy of these variables. Our code augmenter converts these references into indirect references into the executing thread's context block which contains the thread's local state: register values, local variables, and stack.

Dynamic Reference Discrimination: We suggested in Section 2.1.3 that a compiler, with proper language support, should be able to statically identify all memory accesses as going to either local or shared memory. Since we do not have languages and compilers that support this, we have added code augmentation to check address ranges at execution time and determine if a pointer is to shared or local memory. Optionally, this reference classification information can be collected as a trace file on the first run of an application and then fed back into the code modifier to do complete static classification.[2]

Re-Optimization: During our studies of multithreading we found it important to group shared memory load instruction together. We implemented this within the code modifier by reordering instruction and percolating shared memory load instructions up towards the tops of basic blocks.

Extended Instruction Sets: For the most part we accepted the instruction set of the processor on which simulations were being executed: the MIPS R3000[Kan89]. However we did want to add a number of new instructions such as: double word load and stores, local and shared memory versions of all loads and stores, an explicit thread switch instruction, fetch-and-add, and other special synchronization instructions. These were all added by having the code modifier convert these into calls to special simulator routines.

Virtual Registers: On of the most useful new code augmentations is virtualization of the register file. This simplified implementation of the other code augmentations because it eliminated concerns about remapping registers. This will discussed more fully at the end of this section.

B.4.1 An Example

Figure B.2 shows an example of code augmentation for a small code fragment which will be used to demonstrate several of the code augmentations described above. The original assembly language instructions are shown in part (a); the modified code is shown in part(b).[3] These instructions were generated by the compilation of the expression

[2]This accurate static classification is required for our re-optimization of the code.

[3]The instruction set is approximately that of the MIPS R3000[Kan89], but it has been simplified slightly to make the example clearer.

code for:	**A = B + C + X**	
where:	**A**	is variable in shared memory
	B,C	are variables in local memory
	X	is variable in register r8

registers: Rgp = global pointer
Rsbp = shared base pointer
Rcp = context pointer
Rtime = time value

simulator interface:
simulator_sw(r4 = address, r5 = value)

```
lw    r1, local_addr_of_B(Rgp)
lw    r2, local_addr_of_C(Rgp)
add   r3, r1, r8
add   r3, r3, r2
sw    r3, shared_addr_of_A(Rgp)
```

(a) original code

```
lw    r8, offset_of_r8(Rcp)           } load used registers
lw    r1, local_addr_of_B(Rcp)
lw    r2, local_addr_of_C(Rcp)
add   r3, r1, r8
add   r3, r3, r2
sw    r1, offset_of_r1(Rcp)           }
sw    r2, offset_of_r2(Rcp)           } save modified registers
sw    r3, offset_of_r3(Rcp)           }
addi  Rtime, Rtime, 4                 } accumulate time
addi  r4, Rsbp, shared_addr_of_A      }
lw    r5, offset_of_r3(Rcp)           } call out to simulator
addi  Rtime, Rtime, 1                 }
call  simulator_sw                    }
```

(b) modified code

Figure B.2: Example of code augmentation

$A = B + C + X$, where the variables B and C will be loaded from local memory, the variable X is already in register `r8`, and the result A will be stored in shared memory. Assume for this example that this expression by itself forms a basic block. Basic blocks are the granularity at which we perform analysis and code augmentation, and thus this small basic block can serve as a complete example.

The first step is to identify which instructions can be directly executed by the host processor and which instructions will require call-outs to the simulator. In this example the last instruction references shared memory and thus will be replaced with a call-out. The other four instructions are local to the processor and can be directly executed. For ease of manipulation, the call-out instruction is isolated into its own basic block, as indicated by the horizontal lines separating the instructions.

The second step is to calculate the timing of the basic blocks. The first block has four instructions and takes four cycles. The second block has one instruction and takes one cycle[4]. The timing of each basic block is computed statically and is used in the inserted

[4]In general determining accurate timing is somewhat more complicated because the processor pipeline must be modeled. Usually looking just within a basic block is adequate, but sometimes long latency floating point operations continue executing past the end of a basic block and affect the timing of subsequent blocks.

instructions which accumulate the running execution time in register `Rtime`.

The third step is reference indirection. The loads of local variables `B` and `C` are originally relative to the global pointer (register `Rgp`). These are changed to be thread relative by indexing off of the thread context pointer (register `Rcp`).[5]

Step four involves adding code for in-line context switching. In our implementation, we maintain the invariant condition that between basic blocks all register values should be correctly stored in the context block of the executing thread. In our system this context block is pointed to by the `Rcp` register, and thus register load and stores are relative to this pointer.

At the start of each basic block we insert code to load the registers whose values will be used. In the example, only the value in register `r8` is used. The registers `r1`, `r2` and `r3` also appear, but they do not need to be loaded since their original values are not used. At the end of each basic block we append code to store any registers who's values have been redefined. In the example this is `r1`, `r2` and `r3`.

This completes the code augmentation for the first basic block. The second basic block is the save word instruction (`sw`) that originally saved the value in register `r3` to an address in shared memory. It is replaced by a sequence of instructions which load parameters and then call-out to a simulation routine to perform the shared memory operation. The address and data values are loaded into the argument registers (`r4` and `r5`), and the time counter (`Rtime`) is incremented by 1 (the time taken by the original instruction). If the simulator finds that more time would be needed by this instruction, for instance if the memory network is clogged or there is a cache miss, the simulator would add the additional time.

This completes the code augmentation. The code has now been converted so that it is context block relative. The simulator can now switch threads by changing the context pointer and time counter and then jumping into the new thread to be executed.

If these subsequent blocks are selected by conditional branches, the exact timing will depend upon the branch paths taken at execution time. These cases are rare, and for our simulator we use timings based on the statically predicted most likely branch paths.

[5]Here reference indirection is simply changing from `Rgp` to `Rcp` and possibly changing the offset. It is more involved when the original reference is not relative to `Rgp`.

B.4.2 Virtual Registers

The technique of in-line context switching usually leaves most register values in the context block, and this motivated the idea of virtualizing the register file. When register `r8` was loaded and later used in Figure B.2(b), it could have been loaded into any physical register as long as the register later used in the `add` instruction was also changed to the same register. Thus the *virtual* registers used in the original code need not be the same as the *physical* registers used in an expanded basic block. Different basic blocks could choose to use different physical registers to hold the virtual register `r8`.

The benefit of this is that we can now have more virtual registers than there are physical registers. For instance we have used virtual registers `Rtime`, `Rcp`, and `Rsbp` in our modified code. The mapping between virtual and physical registers is possible as long as each individual basic block does not use more virtual registers than there are physical registers to map into. Mapping problems are rare and occur only for large basic blocks, and they are easily handled by splitting these large blocks into multiple smaller blocks.

This virtualization of the register file actually simplifies other code augmentations. For instance in the old style of code augmentation, some specific physical register, say `r30`, is used for time counting. Thus wherever `r30` is used in the original code, the code must be modified to work around the usurpation of this register.

Virtual register have many potential uses. One example use was in a research project that tried to improve memory reference patterns by re-optimizing basic blocks in order to group together shared memory load instructions. This re-optimization needed a few extra temporary registers to allow reordering of instructions while still preserving all data dependencies, and these extra registers were made available as extra virtual registers.

B.5 Performance

In this section we discuss three aspects of the performance of our simulator: the cost of in-line context switching, the slowdown factors of basic simulations, and the affects on slowdown when simulating multithreading or caching.

B.5.1 Cost of In-line Context Switching

Application	Description	Context switch cost: switch in	Context switch cost: switch out	Average interval between switches	Amortized cost per instruction
sieve	finds primes	9.8	7.9	7.0	2.5
blkmat	blocked matrix multiply	47.7	50.3	48.0	2.0
sor	solves Laplace's equation	8.5	5.5	4.2	3.3
ugray	ray tracing renderer	11.8	9.1	10.1	2.1
water	system of water molecules	27.7	22.2	33.1	1.5
locus	standard cell router	8.0	5.2	4.0	3.3
mp3d	rarefied hypersonic flow	8.1	6.3	4.7	3.1

Table B.1: Context Switch Costs

Table B.1 shows the effectiveness of in-line context switching. It gives the context switch frequency and the average context switch costs for the applications that we have used in our simulation studies.

The *switch in* cost listed in the table is the average number of registers loaded per context switch into the application from the simulator. The *switch out* cost is the average number of registers saved per context switch from the application out to the simulator. Recall that these register loads and stores do not all occur at the points of context switching between the simulator and threads, but are spread among the prefixes and suffixes of the sequence of basic blocks executed between context switches. Also included in these context switch costs are the overheads incurred by the simulator in saving and restoring reserved registers such as the program counter, time counter, stack pointer and context pointer.

The column labeled *average interval between switches* shows the average number of simulated cycles between context switches. For those applications that context switch most frequently, the context switch cost is less than 10 cycles. The `locus` program, for example, accesses shared memory very frequently and thus context switches at an average rate of once every four cycles. The average cost of these context switches is 8.0 cycles to switch in and 5.2 cycles to switch out. In all cases, the context switch cost is less than the size of the register set[6]. In comparison, the light-weight thread package used in Proteus[Del91] loads and stores the entire register set and takes 135 cycles per context switch.

In our system, the cost of context switching is roughly proportional to the frequency of occurrence. The longer an application executes, the more registers it is likely to

[6]On a Mips processor there are 29 integer, 32 floating point and 3 special purpose registers in the usable register set.

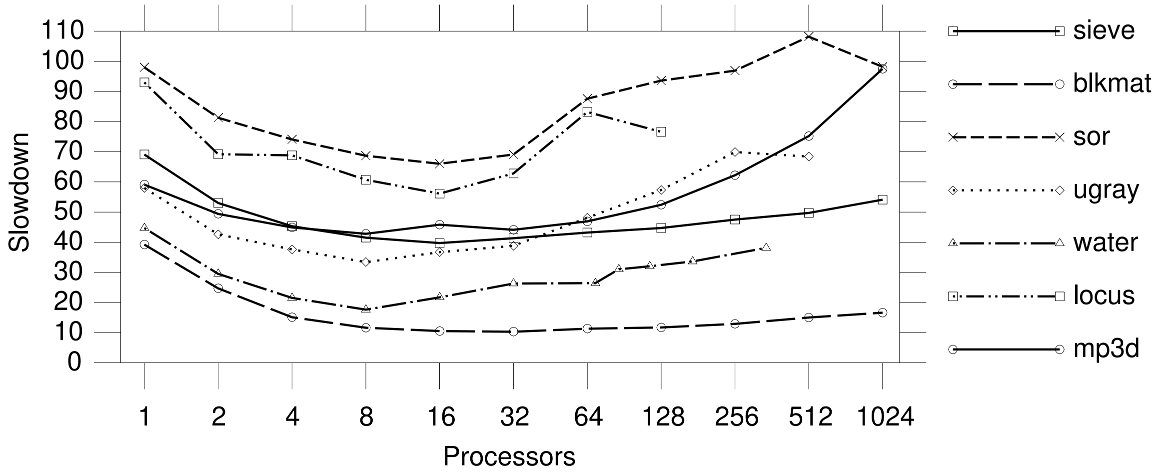

Figure B.3: Simulation slowdown

use. The `blkmat` and `water` applications, for example, context switch less frequently than the other applications and their average context switch costs are higher. However since they do not context switch as frequently, the higher context switch costs are amortized over a longer period. Overall, the total context switch overhead ranges from 2 to 3 cycles per simulated cycle.

B.5.2 Slowdowns Factors for Basic Simulations

Figure B.3 shows the performance of the FAST simulator on the various benchmark applications. Results are shown with the number of processors varied from 1 to 1024. The slowdown factors shown in this graph are the number of cycles taken to simulate a single cycle of a single thread. Since most instructions are directly executed and the context switching cost has been reduced to just 2 to 3 cycles per simulated cycle, one might expect slowdown factors of 3 or 4. The slowdowns are larger because of the remaining overhead which comes from the scheduling mechanism within the simulator, the simulation of shared references, the memory simulator, and statistics gathering. For this graph the memory model is a simple ideal memory that has 0 latency and no contention.

Two interesting trends can be observed from this graph. First, the slowdowns vary for different programs. Programs such as `blkmat` and `water` have typical slowdowns from 10 to 30, while programs such as `locus` and `sor` have typical slowdowns from 60 to 100. The difference comes from the different frequencies at which the applications interact with

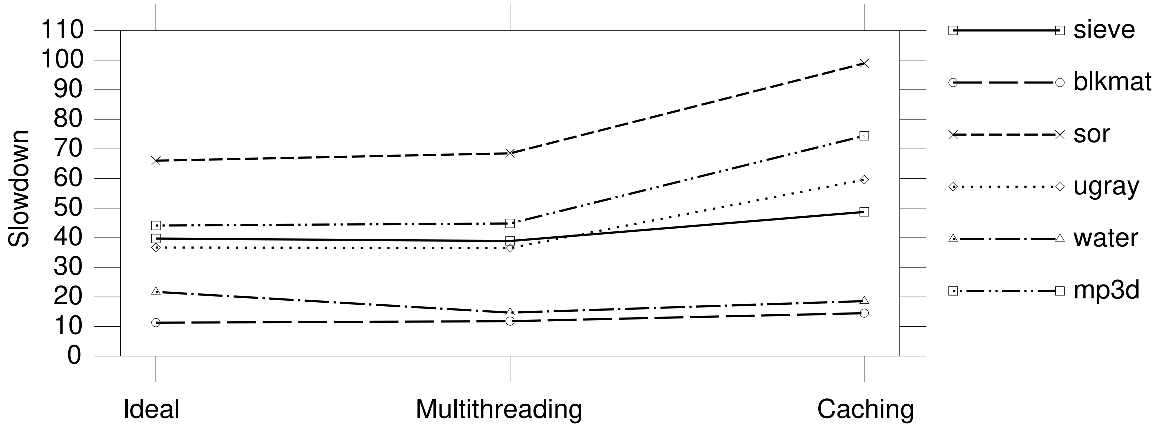

Figure B.4: Simulation slowdowns under different configurations.

the simulator. `Sor` and `locus` had context switches every 4 cycles compared to `blkmat` and `water` which have context switches only every 30 to 50 cycles and thus require much less scheduling by the simulator. The cost of simulated events is amortized over a larger number of instructions, and thus the overall slowdown factors for `blkmat` and `water` are lower than those for the other applications.

The second interesting trend is that as the number of processors is increased, the slowdown factor initially drops and then slowly rises. The initial decrease in slowdown is due to the time wheel algorithm used to schedule threads and events. It works best when there are many processors and thus there are many events per cycle. The later increase in the slowdown factor occurs because the applications use more synchronization operations as the number of processors is increased. Synchronization operations, especially spinning on locks or barriers, involve many shared accesses and thus increase the work of the simulator.

B.5.3 Multithreading and Caching

FAST was designed in a modular fashion and can be configured to perform a wide variety of different simulations depending upon what is of interest to the researcher conducting the simulation studies. The main uses of the simulator have been for studies of multithreading under long memory latencies and for performance studies of cache coherency protocols.

Figure B.4 shows the performance of the simulator under three configurations: the *ideal* case which has 0 latency, the *multithreading* case which has 200 cycle latency and several threads per processor, and the *caching* case which uses a cache simulator of the Censier and Feautrier[CF78] directory based cache coherence protocol. The ideal case and the multithreading case have roughly the same performance. This occurs because studying multithreading was one of the primary intended uses of FAST, and thus multithreading support was built in from the start. Single threaded execution is simply a special case of multithreading in which there is just one thread per processor. The cache simulator typically takes hundreds of cycles per reference to check and manipulate the caches' states, and this extra overhead slows the simulations. The change in performance is moderated by the fact that the cache simulation cost is amortized over the total number of simulated cycles.

B.6 Summary and Future Research

We have used FAST to perform a large number of architectural simulations. Its fast speed has allowed us to simulate larger problems and larger machines than would have been possible with previous comparable simulators. Execution driven simulation is the most important technique for obtaining high performance.

However speed is just one important aspect of FAST. By carefully understanding the tradeoffs in design choices, we have been able to build a simulator that is also more accurate than previous instruction level simulators. The most important point is that code augmentation must be applied at a low level since source code alterations can perturb the object code produced and thus the accuracy of instruction level timings. A second point is that accurate interleaving of global events requires frequent context switching between simulated processors, and thus fast context switching is desirable.

In building FAST, we have extended the idea of code augmentation into a number of new areas such as in-line context switching, re-optimization, extended instruction sets, and virtualization of the register file. These extensions have been important in making the right design tradeoffs so as to obtain both high performance and high accuracy, and in making a simulator that is flexible enough to be used for a large variety of experiments.

There are several possible directions for future research with FAST or similar simulators. First, since we are simulating a shared memory multiprocessor, it should be possible to speed up the simulator be executing it in parallel on today's small shared memory multiprocessors in order to simulate tomorrow's larger machines. The main problem that will arise is synchronizing and correctly interleaving the concurrent simulations of multiple processors.

Second, FAST would be a good foundation for a parallel program development and debugging system. Simulators are useful for debugging because they can reproduce identical timing races on subsequent runs. The Proteus[Del91] simulator provides a powerful monitoring facility by inserting monitoring code into the source code of applications, and we would like to see if similar mechanisms could be built without modifying the applications' source code.

Third, our new augmentation techniques of virtualizing the register file and extending the instruction set could be used along with a modified compiler to study various architectural changes such as larger register files or new instructions.