Fast Accurate Simulation of Large Shared Memory Multiprocessors (revised version)

Bob Boothe*

Report No. UCB/CSD-93-752

June 1993

Computer Science Division (EECS)
University of California
Berkeley, CA 94720

Abstract

Fast computer simulation is an essential tool in the design of large parallel computers. Our Fast Accurate Simulation Tool, FAST, is able to accurately simulate large shared memory multiprocessors and their execution of parallel applications at simulation speeds that are one to two orders of magnitude faster than previous comparable simulators. The key ideas involve execution driven simulation techniques that modify the object code of the application program being studied. This produces an augmented version of the code that is directly executed and performs much of of the work of the simulation. We extend the previous work by introducing several new uses of code augmentation.

In this paper we summarize the tradeoffs made in the designs of this and previous simulators. In previous simulators, these tradeoffs have often led to sacrificing accuracy for faster simulation. However by careful selection of techniques and when to apply them, we have built a simulator that is both faster and more accurate than previous simulation systems. The

*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.

improved accuracy comes from applying code augmentation techniques at a uniform low level and from having such fast context switching that accuracy/performance tradeoffs become unnecessary.

Our simulator has a modular design and has been configured in many ways. It has been used to conduct numerous experiments on multi-threaded machine behavior, application behavior, cache behavior, compiler optimization, and traffic patterns. Because of its high performance, we have been able to perform simulations of larger machines than would otherwise have been feasible.

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[2, 8, 13]. 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[12] or Tango[6] 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*[5] 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[10]. 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.

1.1 Overview

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

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.

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[12], which we are more familiar with since this was done at Berkeley, and ASIM(refer to the description in [7]) 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.

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[6] 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[3, 7] 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 what would have been produced if the original code were compiled directly. 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.

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.

[1]Most of our applications were originally written for the Sequent[14]. 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.

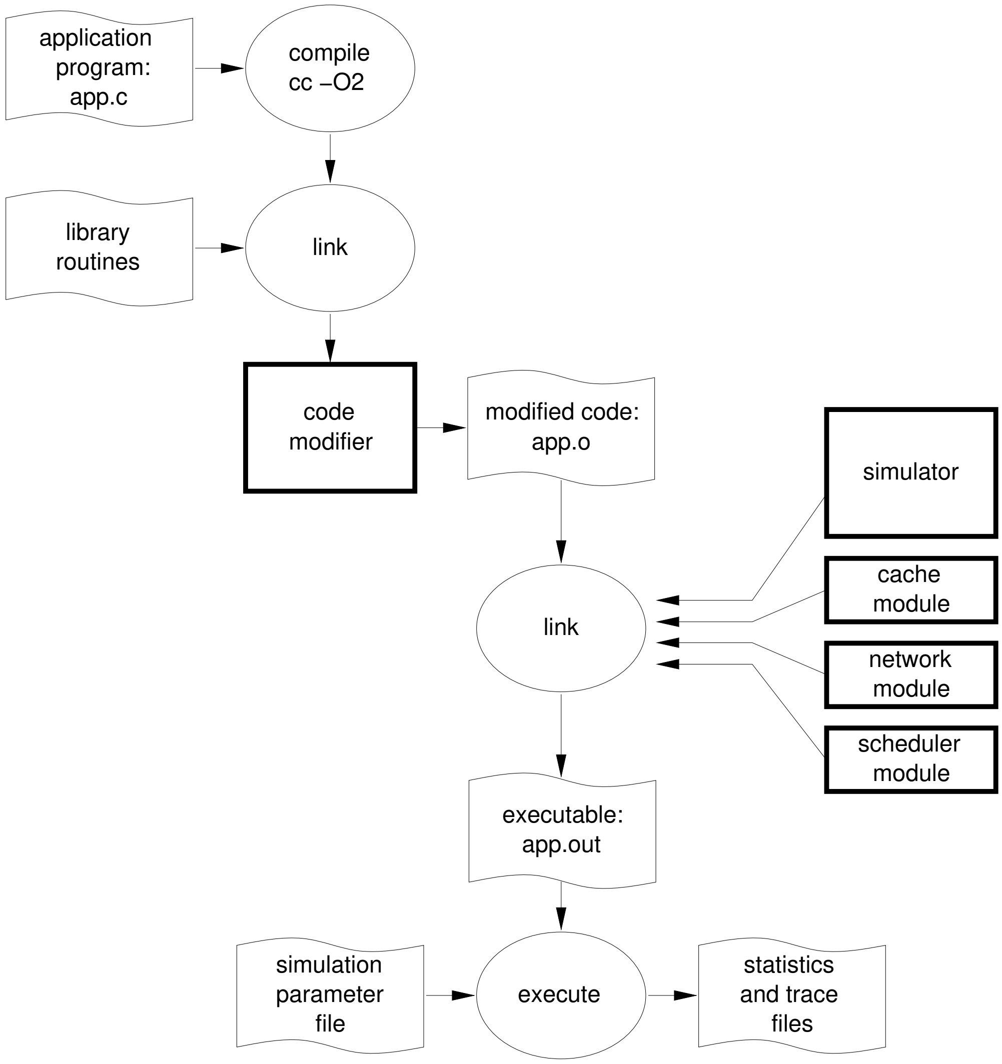


Figure 1: Diagram of using FAST.

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.

3 Simulator

Figure 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 we 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, and 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.

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[11].

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.

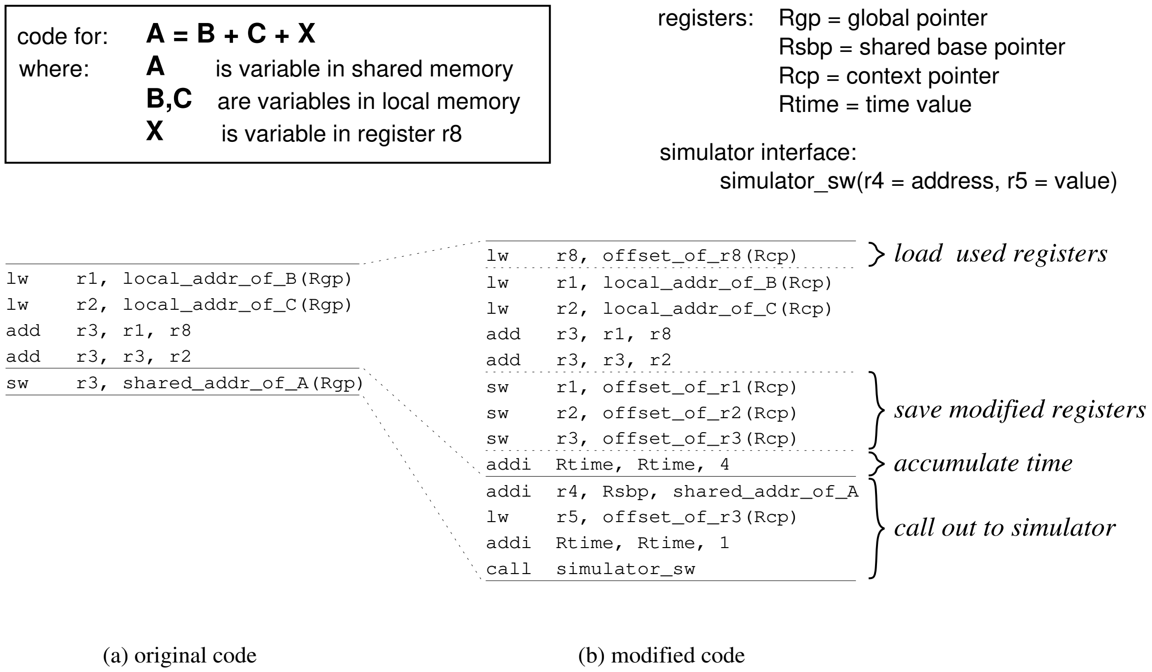
Reference Indirection: For a single threaded program, which the compiler thinks it is compiling, static local variables are assigned to fixed memory addresses. However for a parallel program each thread needs its own copy of these variables. Our code augmenter converts these references into indirect 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 have suggested that with proper language support[2], a compiler should be able to statically identify all memory accesses as going to either local or shared memory. Since we don't 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[9]. 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.

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



(a) original code (b) modified code

Figure 2: Example of code augmentation

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

4.1 An Example

Figure 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 $A = B + C + X$, 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

[3] The instruction set is approximately that of the MIPS R3000[9], but it has been simplified slightly to make the example clearer.

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 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 aren't 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.

[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. 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`.

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 2(b), it could have been loaded into any physical register as long as the later use in the `add` instruction was also changed to use 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 doesn't 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.

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.

5.1 Cost of In-line Context Switching

Table 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. `Sieve`, `blkmat`, and `sor` are toy applications developed by the author. `Ugray` is from Berkeley[1]. `Water`, `locus`, and `mp3d` are from the Stanford SPLASH[15] benchmark set.

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

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 1: Context Switch Costs

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[7] 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, because the longer an application executes, the more registers it is likely to 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 don't 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.

5.2 Slowdowns Factors for Basic Simulations

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

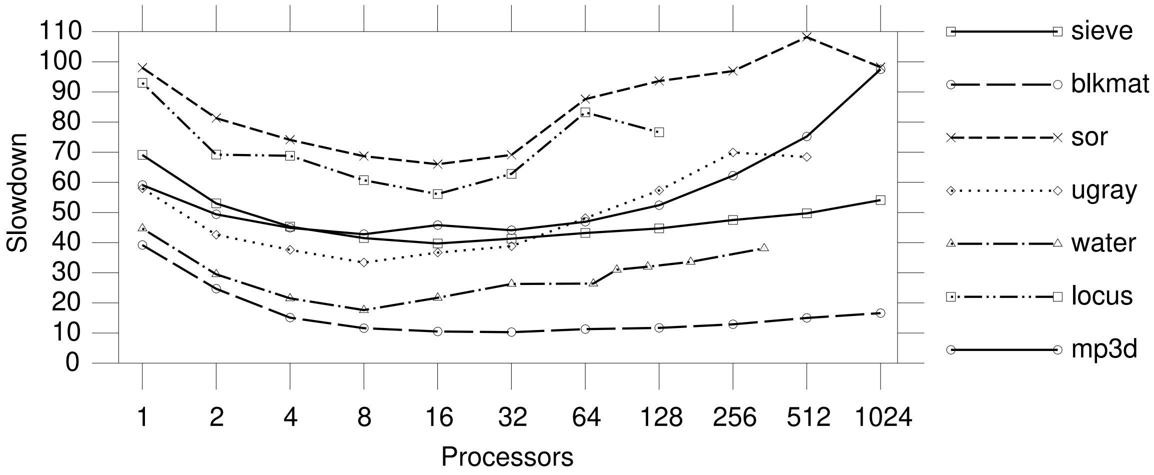


Figure 3: Simulation Slowdown

Figure 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 iss 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 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.

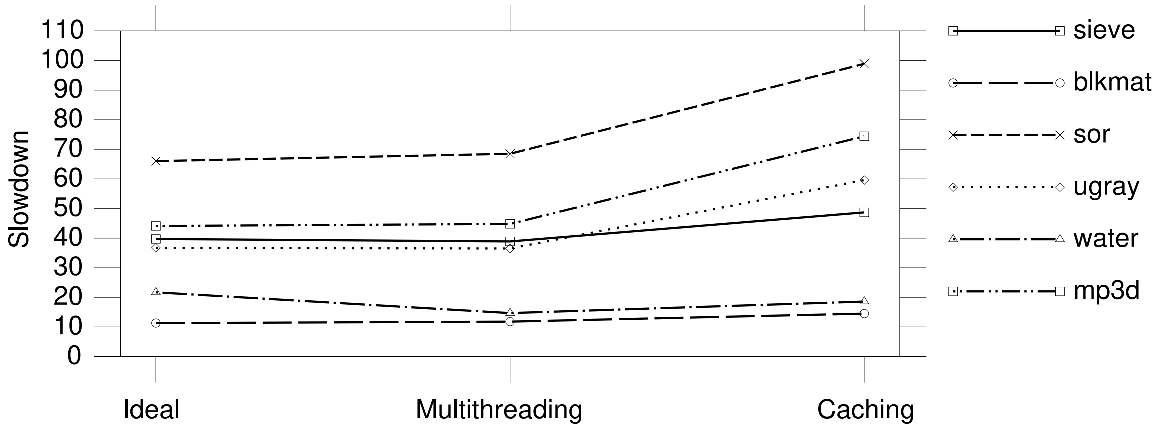


Figure 4: Simulation slowdowns under different configurations.

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 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[4] 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.

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 extension 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 multiprocessors, it should be possible to speedup 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[7] 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 machines with larger register files or the utility of new instructions.

References

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

[2] 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.

[3] 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.

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

[6] 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.

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

[8] 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.

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

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

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

[12] 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.

[13] 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.

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

[15] 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.