View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Southern Maine: Digital Commons@USM

University of Southern Maine

USM Digital Commons

Faculty Publications | Computer Science

1994

Fast Accurate Simulation of Large Shared Memory Multiprocessors

Bob Boothe PhD
University of Southern Maine, boothe@maine.edu

Follow this and additional works at: https://digitalcommons.usm.maine.edu/computer-science-papers

Part of the Computer Sciences Commons

Recommended Citation

Boothe, B. (1994). Fast accurate simulation of large shared memory multiprocessors. In Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences, Wailea, HI, (pp. 251-260), doi: 10.1109/HICSS.1994.323166.

This Conference Proceeding is brought to you for free and open access by the Computer Science at USM Digital Commons. It has been accepted for inclusion in Faculty Publications by an authorized administrator of USM Digital Commons. For more information, please contact jessica.c.hovey@maine.edu.

Fast Accurate Simulation of Large Shared Memory Multiprocessors

Bob Boothe*

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
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 at simulation speeds that are one to two orders of magnitude faster than 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 does much of the work of the simulation. In particular, we introduce a technique for in-line context switching which reduces the typical context switch time to 10 cycles or less. This fast context switching is an essential component of a shared memory simulator because the simulator must rapidly switch between threads in order to accurately interleave the shared memory references.

*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.

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. For example, simulating the execution of a one MIP uni-processor for one second requires simulating one million instructions, but simulating the execution of a parallel machine with a thousand similar processors for one second requires simulating one billion instructions. Most simulation based research is limited in scope and accuracy by the speed of their simulators[2, 8, 12]. 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 factor[1] ranging from 10 to 100. The slowdown 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 Tango[6] have reported slowdowns ranging from 500 to 6000 for simulations of comparable accuracy.

FAST was developed for the purpose of studying large shared memory multiprocessors with hundreds or even thousands of processors. On these machines we expect that the shared memory will be reached through multistage switching networks which will have delays of hundreds of cycles before results are returned. We are currently studying multithreaded processors as a means of tolerating these long memory latencies. In a multithreaded processor several active threads are maintained and the processor rapidly context switches among them to avoid stalling on memory accesses. 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 an 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 simulate each instruction individually[10]. The assembly code of the application program is augmented to allow the application to guide its own simulation. This augmentation includes adding code to count its own simulated time and code to interact with 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.

[1]The simulation slowdown factor is the number of cycles it takes to simulate a single cycle of execution for a single processor.

One of the main costs in an execution driven simulation of a parallel machine is the overhead for context switching between the many parallel threads in the application program. To accurately simulate a shared memory machine, the memory access of all of the parallel threads must be correctly interleaved. This requires context switches at every reference to shared memory. For some applications, as many as one in five instructions may be a shared reference and thus context switching will occur every fifth simulated cycle. In fact, each reference actually causes two context switches: one context switch to the simulator, and later a context switch back to application. For many applications, the context switch cost is a critical factor in the performance of the simulator.

In this paper we show how to greatly reduce the cost of context switching by introducing a technique for in-line context switching. This extends the ideas of execution driven simulation with code augmentation that lets the simulated program manage its own register set. Without this technique, a context switch involves saving or restoring the full set of 64 registers.[2] Often only a few of these registers are actually used before the next context switch, and it is wasteful to save and restore the entire set. Our technique reduces the number of loaded registers to only those registers whose value are used, and it reduces the number of saved registers to only those registers whose values have changed. The typical context switch time is reduced to less than 10 cycles.

The remainder of this paper is broken into four sections. In section 2 we explain and demonstrate the techniques of code augmentation. In section 3 we discuss the overall simulation system and its performance. In section 4 we compare our performance to other simulators. And in section 5 we conclude.

2 Code Augmentation

Code augmentation is the process of inserting extra code into an application program. The inserted code can perform various functions such as keeping track of the simulated execution time or gathering statistics. The MIPS `pixie` program[11], for instance, collects profiling statistics using code augmentation techniques. In this section we show how to extend the code augmentation technique with the ability to support rapid context switching.

Although code augmentation can be performed on a high level language[5], it is best performed at the assembly level for several reasons. First, compiler optimizations will have already been performed and thus the application program can be studied in the optimized form in which it would be run on a real parallel machine. Second, the timing of the application can be accurately determined from the assembly language instructions[3]. Finally, the assembly language instructions are easily manipulated and

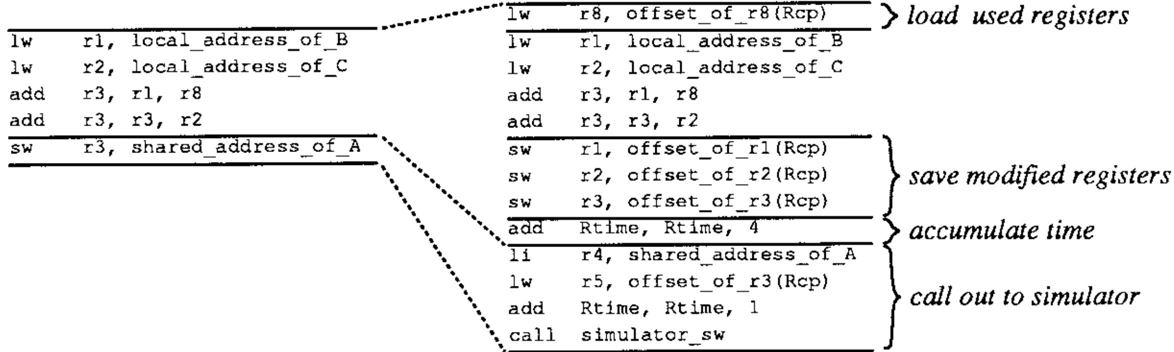
[2] On a Mips processor there are 29 integer, 32 floating point and 3 special purpose registers in the usable register set.

[3] This is true for RISC instructions since timing and pipeline delays are easily determined. A

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: Rcp = context pointer
Rtime = time value

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



(a) original code

(b) modified code

Figure 1: Example of code augmentation

reorganized. We will now present a simple example which shows the steps in code augmentation.

2.1 Example

Figure 1 shows an example of code augmentation for a small code fragment. The original assembly language instructions are shown in figure 1(a).[4] 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 must be simulated. In this example the last instruction references shared memory and will be simulated while the other four instruction are local to the processor and can be directly executed. The simulated instruction is isolated into its own basic block and treated separately.

CISC instruction set might be more complicated.

[4]The instruction set is approximately that of the MIPS R2000[9], but it has been simplified slightly to make the example clearer.

We now have two basic blocks as indicated by the lines separating the instructions. The first basic block is directly executed and will be augmented with code that accumulates the timing of the block and with code that saves and restores appropriate registers. The second basic block contains the instruction that needs to be simulated, and it will be replaced with a sequence of instructions that load needed arguments and then call the simulator.

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[5]. The timing of each basic block is computed statically and will be used later during code augmentation. At that point the application will be modified to compute its own simulated execution time by summing basic block execution times as it runs.

Once the execution times of basic blocks have been computed, we are ready to start augmenting the code. Step three involves adding code to manage the register file. As mentioned in the introduction, calls to the simulator are context switch points and the register set must be saved in memory at these points. Since context switches occur frequently and only a few registers are typically used between context switches, it is much more efficient to load only those registers that will be used before the next context switch, and to save only those registers that have changed since the previous context switch. For typical applications, context switches occur almost every basic block, and thus a sufficient policy is to load from and save to the register set at basic block boundaries. We will maintain the condition that between basic blocks all registers values will always reside in memory.

Figure 1(b) shows the expanded code. Before the first basic block we have added a block which loads the registers whose values are used in that block. In this example, only register `r8` is loaded. The registers `r1`, `r2` and `r3` also appear in the basic block, but they do not need to be loaded since they are defined within the basic block and their previous values are not used. At the end of the basic block is append code to save any registers which were defined within the basic block. The registers `r1`, `r2` and `r3` were defined and thus there are instructions to store each of them back to memory. The register values are saved and restored from a region of memory called a context block. In a parallel program, each thread has its own context block, and the registers are saved and restored relative to the context block of the executing thread. For our system, the current context block is always pointed to by a reserved register `Rcp` (context pointer) which contains the starting address of the context block.

The fourth step is to add an instruction to accumulate the time taken by the basic block. Since previous analysis had calculated that the first basic block took four cycles, we append an instruction which adds four to the time counter. The reserved register `Rtime` holds the time value and its value is used by the simulator to order the scheduling of threads.

[5]In general determining accurate timing is somewhat more complicated because of pipeline conflicts within the floating point unit.

Application	Description	Cycles / Switch	Context switch cost	
			Switch in	Switch out
sieve	finds primes	7.0	9.8	7.9
blkmat	blocked matrix multiply	48.0	47.7	50.3
sor	solves Laplace's equation	4.2	8.5	5.5
ugray	ray tracing renderer	10.1	11.8	9.1
water	system of water molecules	33.1	27.7	22.2
locus	standard cell router	4.0	8.0	5.2
mp3d	rarefied hypersonic flow	4.7	8.1	6.3

Table 1: Context Switch Costs

This completes the code augmentation for those basic blocks that are directly executed. Now in step five we replace the instructions that interact with the simulator. These instructions were each isolated into their own basic blocks earlier. The save word instruction (`sw`) that originally saved the value in register `r3` directly to an address in shared memory is replaced by a sequence of instructions which call 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, it would add the additional time.

This completes the code augmentation. The simulator can now use this modified code to simulate parallel programs. By maintaining separate context blocks, program counters, stacks, and time counters for each thread, a large parallel processor can be simulated.

Table 1 gives 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[13] 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 the average number of registers saved per context switch from the application out to the simulator. Included in these 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.

This table shows the effectiveness of this in-line context switching technique. The column labeled *cycles / switch* shows the average number of simulated execution cycles between context switches. For all applications, the context switch cost is less than the size of the register file, and for the 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 context switches at an average

rate of once every four cycles.

The `blkmat` and `water` applications context switch much less frequently than the other applications. Their average context switch cost is higher, but since they don't context switch as frequently, the higher context switch cost is amortized over a longer period. Overall, the total context switch overhead ranges from 2 to 3 cycles per simulated cycle.

2.2 Virtual Registers

The technique just presented of keeping the register values in memory facilitates simple virtualization of the register file. For example, when register `r8` was loaded and later used in figure 1(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`.

Some virtualization of the register file is necessary to do code augmentation because a few of the registers used by the original program have been usurped for special purposes. Examples are the `Rtime` and `Rcp` registers used in the modified code to hold the time counter and context pointer respectively. These extra register are assigned to arbitrary physical registers (in our case `r23` and `r30`). Wherever `r23` and `r30` are used in the original code, they must be remapped to use some other register.

This capacity for virtual registers has proven very convenient in later research projects performed with this simulator. In particular a research project involving reorganizing the basic blocks to improve the reference patterns to shared memory needed a few extra temporary registers to allow reordering of instructions while still preserving all data dependencies. These extra registers were made available as extra virtual registers, and then mapped into physical registers on a block by block basis.

3 Simulator

The simulator itself is written in a high level language (C) and is easily configured. It can simulate different multithreading models and can use different memory system models. For studies of multithreading it was used with a simple memory model with constant memory access latency. For studies of cacheing it was configured with memory modules that simulated an assortment of cache coherence protocols.

The main activity of the simulator is scheduling the threads and the memory operations based on their time ordering. The simulator also gathers various execution statistics and provides special system routines, such as `m_fork()`, which allow the application to create and manipulate the simulated parallel threads.

Figure 2 shows the performance of the simulator over a large set of applications. Results are shown with the number of processors varied from 1 to 1024. The slowdown

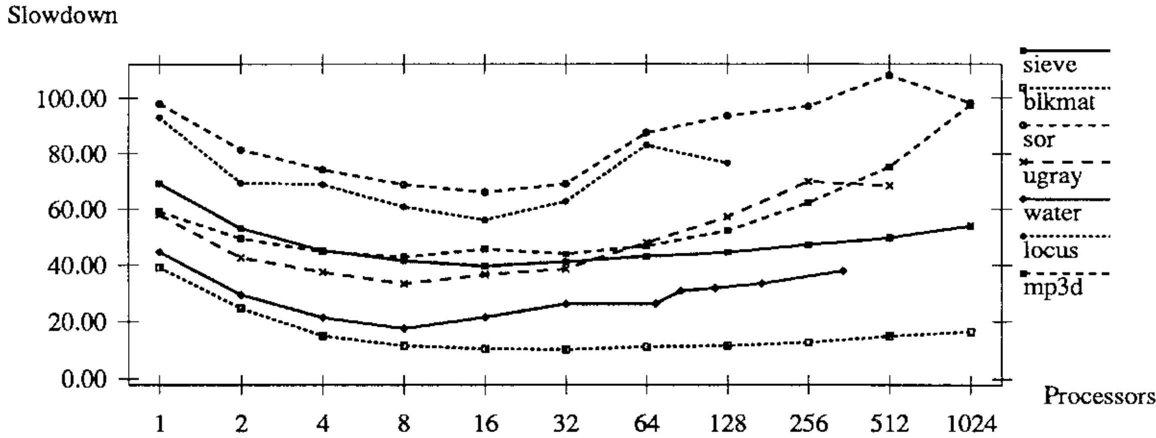


Figure 2: Simulation Slowdown

factors shown in this graph are the number of cycles which are 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 much 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 was 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. Where `sor` and `locus` had context switches every 4 cycles, `blkmat` and `water` have context switches only every 30 to 50 cycles and thus require must less scheduling by the simulator. The costs of scheduling operations are amortized over a larger number of simulated 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 simulated 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 implement the priority queue that is used for scheduling threads and memory accesses. 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.

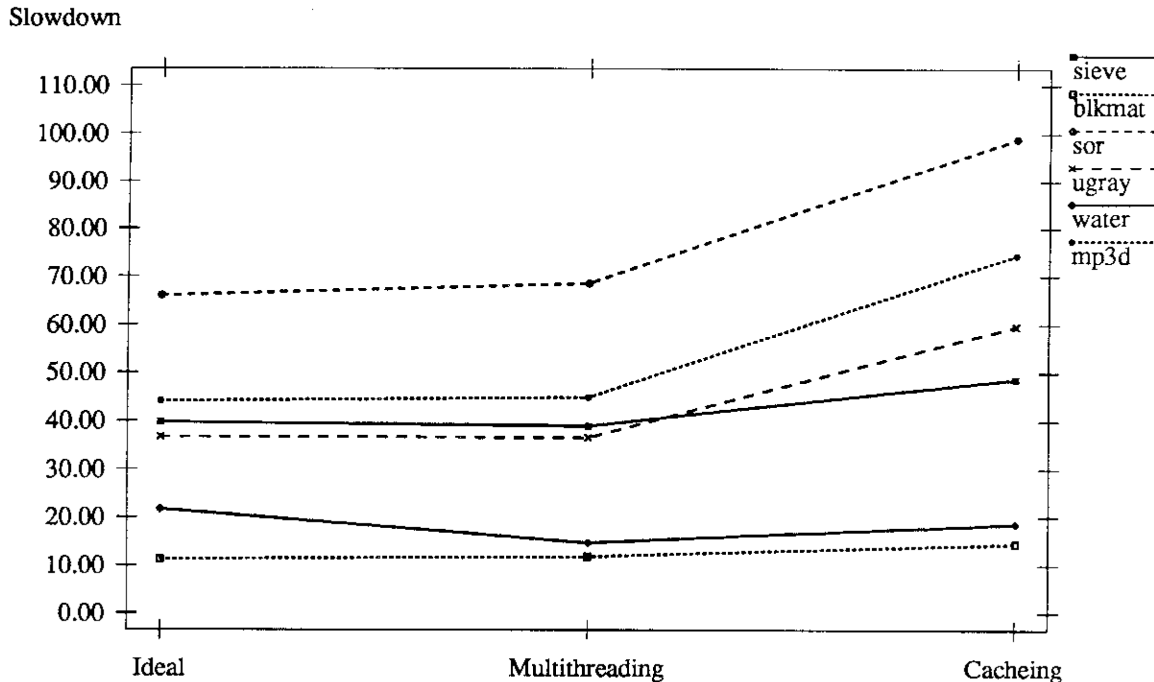


Figure 3: Simulation slowdowns under different configurations

3.1 Memory Simulator

This simulator can be combined with an assortment of memory simulators. The different memory simulators are used 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 3 shows the performance of the simulator under three configurations. The configurations are: the *ideal* case which has 0 latency, the *multithreading* case which has 200 cycle latency and anywhere from 3 to 11 threads per processor depending on the application, and the *cacheing* case which uses a memory simulator that simulates the Censier and Feautrier[4] directory based cache coherence protocol. The ideal case and the multithreading case have roughly the same performance. This is because studying multithreading was our main use of the simulator, and thus the multithreading support was built into the simulator from the start. Single threaded execution is simply a special case of multithreading in which there is just one thread per processor. The *cache* simulations use a memory simulator that typically takes hundreds of cycles per reference to check and manipulate the caches' states. This extra overhead for the memory simulator slows down the simulations, but the slow downs are only moderate because the memory simulation cost is amortized over the

total number of simulated cycles.

4 Related Work

The Tango simulator[6] developed at Stanford is based on Unix shared memory and uses Unix context switches. Unix context switches take tens of thousands of cycles and slow down their simulator tremendously. For accurate simulations they report slowdown factors ranging from 500 to 6000. In compensation for their large context switch cost, they provide a mechanism to allow faster simulation in exchange for reduced accuracy of the results. The faster simulation lets a thread execute hundreds of memory references before it is context switched. The loss in accuracy arises because the memory references of the threads are no longer accurately interleaved.

They are currently converting their simulator to use a light-weight threads package. This will greatly reduce the cost of context switches and will bring their performance closer to the performance of our simulator.

Recently we learned of the Proteus simulator developed at MIT[3, 7]. Their simulator uses execution driven simulation and a light-weight threads package. This is a versatile simulation system allowing simulation of both message passing and shared memory multiprocessors. Their light-weight threads package takes 135 cycles per context switch, compared with the typical context switch cost on our system of less than 10 cycles. Their performance, however, is fairly close to that of our simulator since there are other large costs such as the ordering and scheduling of events and the simulation of shared memory accesses. They have reported typical slowdown factors ranging from 35 to 100.

5 Conclusions

We have used our FAST simulator to perform a large number of architectural simulations. Its fast speed has allowed us to simulate larger problems and larger machines than would be possible with previous comparable simulators. Simulations that can be completed in an hour on our simulator would require days on slower simulators.

The performance of our simulator comes from the combination of execution driven simulation techniques and our new technique of in-line context switching. Where previous execution driven simulators have used Unix processes where context switching costs 10,000+ cycles or light-weight threads where context switching costs 100+ cycles, we have reduced the typical context switch cost to less than 10 cycles.

This large reduction in cost is made possible because context switches occur so frequently that typically only a few registers are used between context switches. The assembly code can be augmented to load and save only those registers which are used. This reduces context switch overheads to just 2 or 3 cycles per simulated cycle.

The most critical need for fast context switching occurs when context switching occurs most frequently. The in-line context switching approach performs well in this situation since as the interval between context switches decreases, the context switch cost decreases as well.

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*, 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 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.

[13] 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.