Statistical Simulation of Symmetric Multiprocessor Systems

Sebastien Nussbaum

Sun Microsystems, Inc.
Chelmsford, MA 01824
sebastien.nussbaum@sun.com

James E. Smith

Dept. of Electrical and Computer Engineering
University of Wisconsin – Madison
Madison, WI 53706
jes@ece.wisc.edu

Abstract

Statistical simulation is driven by a stream of randomly generated instructions, based on statistics collected during a single detailed simulation. This method can give accurate performance estimates within minutes, allowing a large design space to be simulated quickly. Prior work has applied this technique to superscalar processors. We evaluate the extension of statistical simulation to Symmetric Multiprocessing (SMP) systems. Key program parameters are identified, and program statistics are collected during detailed simulations for both multiprogrammed workloads (SpecInt) and parallel scientific workload (Splash-2). The accuracy of statistical simulation is evaluated at different levels of model detail, and it is shown that for multiprogrammed workloads a 10% average error can be achieved, and for parallel benchmark programs 15% average error can be achieved.

1. Introduction

For uniprocessor systems, optimizing an architecture often requires heavy use of simulation tools to evaluate the performance of a number of different hardware configurations (i.e. reorder buffer size, issue width). Simulation times are many orders of magnitude slower than real time, making it difficult to evaluate numerous design alternatives. Recently, statistical simulation [1,2,3,4] has been proposed as an efficient method for modeling uniprocessor performance. After an initial detailed simulation, during which program statistics are collected, it is possible to evaluate processor design variations two to three orders of magnitude faster than with conventional detailed simulation, while only losing a few percent accuracy. Consequently, statistical simulation is a useful technique for narrowing the design space during the early phases of design.

For multiprocessor systems the simulation problem is much worse than for uniprocessors. Not only are multiprocessor systems much larger in scale, but they also have additional design choices. Furthermore, problem sizes tend to be larger. In this paper, we study the extension of statistical simulation to symmetric multiprocessor (SMP) systems. We first consider multiprogrammed workloads, for which the performance estimation technique is very similar to uniprocessor systems. We then evaluate statistical simulation for parallel scientific workloads, which differ from multiprogrammed workloads by the presence of software synchronization and accesses to shared memory pages.

In Section 2, we present the important aspects of statistical simulation as applied to superscalar processors. This includes instruction, cache, and branch prediction statistics used for modeling a single thread of execution. In section 3 the multiprocessor architecture we have chosen for initial study is described. In Section 4, we describe the additional set of statistics to collected for accurately modeling synchronization and cache coherence events in multiprocessor systems. Section 5 presents simulation methodology and the accuracy of statistical simulation is compared with detailed multiprocessor simulation, first for multi-programmed workloads, then for parallel workloads.

2. Statistical Simulation

The overall method we propose is to first develop a statistical model for individual uniprocessors, then combine them, along with interconnect and memory components, into a statistical multiprocessor model. The uniprocessor model is similar to the one developed in [4], enhanced with additional features necessary for modeling interprocess synchronization and communication.

2.1. Uniprocessor modeling

Statistical simulation is a two-step process as shown in Figure 1. First, a benchmark program is simulated with a detailed execution driven simulator to produce a dynamic instruction trace. The program instruction trace is

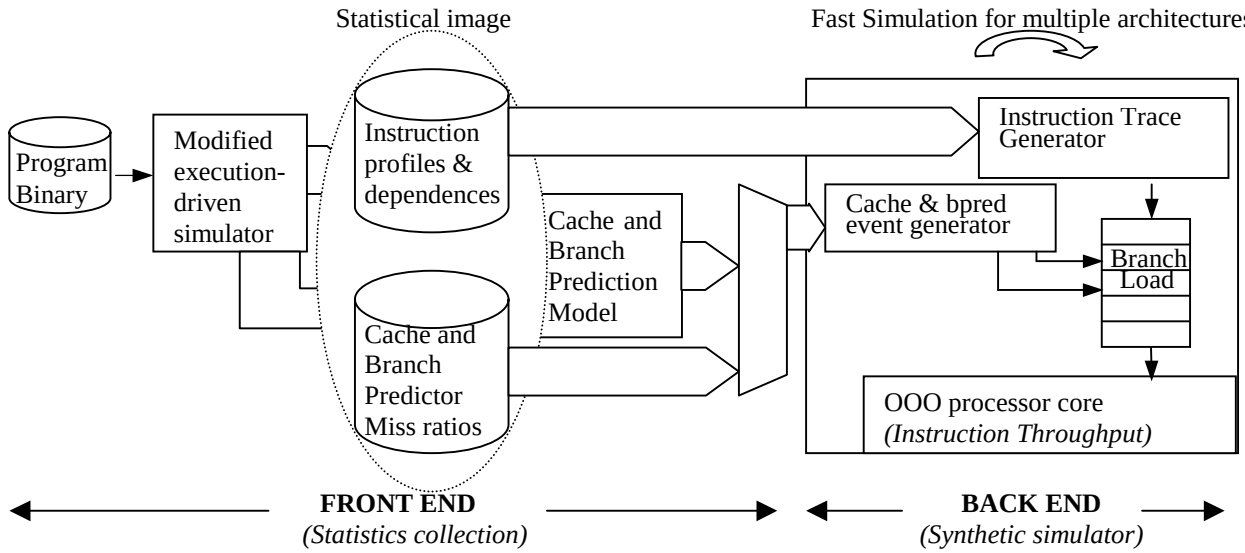

Figure 1: Statistical simulation process flow

then analyzed to generate statistical tables of key program characteristics such as instruction profiles, register dependences, and cache/branch predictor performance. These statistics represent the program behavior during its execution on one particular input. This task only needs to be done once for each benchmark, and creates the program's *statistical image*.

To carry out a statistical simulation, the tables are used to drive the random generation of a synthetic instruction trace. Branch prediction and cache statistics are then applied to the synthetic trace. This synthetic instruction trace contains all information necessary to compute the timing information for each instruction and is fed into an out-of-order processor simulator. The processor model is relatively simple, modeling the major pipeline resources without ever having to compute values, store results, or maintain register rename arrays. Using the randomly generated instruction stream, the statistical simulation quickly converges (e.g. within 10K to 100K cycles) to a performance estimate typically within 5% error when compared to detailed simulation [4].

2.2. Program statistics

2.2.1. Opcode Mixes

Distinguishing major classes of instructions which use the same resource(s) and have the same latencies is sufficient for achieving accurate timing simulation. Therefore we maintain distributions of 13 instruction types for uniprocessor programs [4], although fewer could probably be used. Section 4 introduces three additional instruction types for multiprocessor systems. In earlier work [4] we identified several methods for modeling instruction distributions. In this paper we focus on the simplest, the *Global Mix*, which contains the relative proportion of each opcode type in the program trace. Other, more advanced mixes used in [4] keep separate distributions for basic block sizes, different mixes for each basic block size, and even different mixes depending on the distance from the beginning the basic block. Such detail is important for reducing error in uniprocessor simulations where very high levels of accuracy may be required. For multiprocessor simulations, however, we have found that in most cases the simpler *Global Mix* provides the best tradeoff between accuracy and simulation time.

2.2.2. Register Dependences

Register dependencies are modeled using a dependence table containing the probabilities that an instruction at distance *d* in the future will be dependent on the instruction currently being generated. Separate statistics are kept for each synthetic opcode type. Dependence probabilities are kept for all instructions up to *N* in the future, where *N* is the maximum issue window size (e.g. 64 or 128). For instructions farther apart than the window size, the dependence should not affect performance. Figure 2 shows the dependence distribution for integer alu (IntALU) instructions in benchmark *Gcc*.

2.2.3. Memory Dependences

If the data input register to a *store* instruction is not available when the store issues, subsequent loads to the same address should stall, waiting for possible forwarding. Similarly, when multiple outstanding loads target the same address, only the first one should be sent to the bus (and main memory), and the others should wait until the data is forwarded to their reservation stations. Therefore, we keep a table of memory dependences similar to register dependences, i.e. the probability that the Nth following memory instruction targets an address overlapping (partially or completely) with the memory instruction currently being generated.

2.3. Locality Statistics

Cache and branch prediction performance are dependent on microarchitecture parameters, unlike instruction mixes and dependences. Because we found no good (and simple) architecture-independent locality capturing model, our statistical simulation is not able to simulate cache or branch prediction performance for a cache or predictor size for which no statistics were collected. This shortcoming may be mitigated if the simulation evaluates processor performance directly for different cache and branch prediction miss rates, instead of structure sizes. In this case, it is possible to collect statistics for two (or more) cache sizes, interpolate the cache miss ratios between the two points, and use statistical simulation to obtain an estimate of the processor performance between these two sample points.

For each of the locality events in Table 1, we keep a probability conditional on the instruction type and previous locality events applied to prevent impossible event combinations.

Table 1. Locality events used during statistical simulation

Event	*Simulator Action*
Branch Target Buffer Miss rate	Stall Fetch until Dispatch
Branch Misprediction rate	Stall Fetch until Writeback
L1 Data Cache Miss rate	6 Cycle operation latency
L1 Data Cache Writeback ratio	Occupies address and data bus
L2 Data Cache Miss rate	32 cycle operation latency
L2 Data Cache Writeback ratio	Occupies address and data bus
Data TLB Miss rate	32 cycle operation latency
L1 Inst. Cache Miss rate	Stop and restart Fetch 6 cycles later
L2 Inst. Cache Miss rate	Stop and restart Fetch 32 cycles later
Inst. TLB Miss rate	Stop and restart Fetch 32 cycles later

3. Modeled SMP Architecture

To demonstrate the application of statistical simulation to multiprocessor systems we consider a particular SMP architecture in detail. The architecture has many of the features (and complications) of today's SMP systems containing superscalar processing elements.

3.1. Memory Architecture

3.1.1. Memory hierarchy

The system we model is built around a split transaction address/data bus. Each processor has separate L1 instruction and data caches, and a unified L2 cache (Figure 3). All caches follow the write-allocate, write-back policy. Main memory may be split into multiple banks, interleaved on a cache block basis. If the target bank is busy, the request is put into a bank queue.

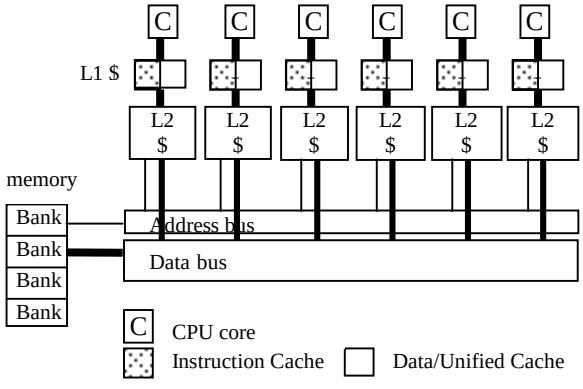

Figure 3: Symmetric multiprocessor architecture modeled

3.1.2. Cache Coherence

We model caches with a MOESI [5] cache coherence protocol. Processors queue for bus ownership. If a bus transaction is a read request, it is forwarded to the memory if the data is owned by memory, or directly queued to the data bus if another cache can supply the data. All instructions triggering a *writeback* event on the second cache level have to queue for the shared address bus first, and then send the cache line to memory via the data bus. Instructions responsible for a writeback do not need to wait for completion of the operation before committing. A *store*, *acquire*, or *release* operation (further described in Section 4.2) hitting in a cache informs other caches of the potential data change (*Invalidation*) by broadcasting the data address on the shared bus before issuing the instruction.

3.1.3. Sequential Consistency

For our study, we assume sequentially consistency, which implies that all memory operations should appear to all processors as if they had executed at the commit stage.

Because we model aggressive speculative superscalar out-of-order processors [6,7], *loads* and *stores* missing in the cache fetch data from memory or the second level cache speculatively, before they commit. However, if a processor sees a bus invalidation for one of the cache lines it has speculatively loaded, it needs to replay the speculative memory instruction, as well as all subsequent instructions.

3.1.4. Shared And Private Virtual Memory Pages

If a processor issues a store to a cache line that it does not own, the store only completes when an invalidation bus transaction is put on the address bus. However, *stores* to shared cache lines cannot be put into a write-buffer because sequential consistency is implemented. Consecutive *stores* to *private* virtual memory pages can issue (and be sent to memory) if all their input registers are ready, if there is a cache port available, if all previous instructions are stores to private pages, and if all of them are ready to issue. For example, with two general cache ports, a sequence of four stores to private pages can be sent to memory at the rate of two per cycle, even if all the stores miss in the second level cache. On the other hand, if the four stores access shared pages, each *store* can only be sent to memory if the invalidation for the preceding store has reached the address bus.

3.2. Processor Synchronization

The benchmark programs we model use both critical sections and barrier synchronizations to coordinate communication and sharing of data. For critical sections, the architecture implements *load-linked* and *store-conditional* instructions. The *load-linked* instruction is retried until it finds the lock variable is clear (unlocked). It then *acquires* the lock by via a *store-conditional* to the same memory location. If the memory location has been invalidated since the *load-linked* completed, the *store-conditiona*l will fail, indicating that another processor entered the critical section first. When the program is finished with a critical section, it simply releases the lock via a conventional store instruction.

4. Multiprocessor Statistics

To model the multiprocessor features described in the previous section, we use statistics for modeling cache coherence events, sequential consistency events, lock accesses, and barrier distributions.

4.1. Cache Coherence and Sequential Consistency

We add to the locality statistics presented in Table 1 the probability that a *store* does not own the cache line it targets. During statistical simulation, if a *store* instruction is randomly marked to access a cache line it does not own, it will not complete until a bus invalidation transaction has reached the address bus.

Additionally, in order to capture the performance increase of successive *store*s to private pages, we keep a distribution of the number of consecutive stores to private pages, and another distribution of the number of consecutive stores to shared pages. The statistical simulator randomly generates a number N of *shared* stores, and will mark the next N *stores* generated as *shared Page Accesses*. It then does the same for private *stores*, and so on.

4.2. Modeling Synchronization

In order to statistically simulate synchronization, we add three new synthetic instructions types: *acquire*, *release*, and *barrier*. Descriptions and usage of these instructions are described in the following subsections.

4.2.1. Lock Distributions

As explained earlier, when a processor successfully executes a *store-conditional* on the lock variable of a critical section, it effectively enters the critical section. During the statistics-gathering phase, we record all the associated instructions leading up to the successful *store-conditional* as a single *acquire* operation, even if the processor retries the lock many times. For each *acquire*, or successful *store-conditional*, we look at the memory address to which it was issued. If this memory address has not been seen on any processor for an earlier *acquire*, we create a new *lock number* for this address. When an *acquire* operation is later issued to this address, the corresponding *lock number* is fetched and updates data being maintained for each *acquired-lock number*. Each processor keeps its own *acquired-lock number* distribution. When the processor exits the critical section it releases the lock by simply storing a zero to the lock address. We record this event as one *release* operation in the statistical architecture.

We maintain two statistical images for each processor: one updated when the processor executes outside any critical section, the other updated when the processor is inside a critical section. During statistical simulation, when a processor reaches an *acquire* operation, the targeted critical section is randomly computed based on the distribution of acquired *lock numbers*, collected during the detailed simulation. The processor will spin on the *acquire* operation, until the section is free (no other processor is executing inside). After entering the critical section, instructions are generated based on the critical statistical image until an *acquire* is generated. Therefore, the average length of a statistical critical section and a critical section from the real execution are the same.

When a processor tries to acquire a lock the first time after other processors successfully enter that critical section, it incurs a cache miss, since the lock content has changed. We model this feature by marking the critical section as a *'future miss'* to all other processors. Likewise, when the processor exits the critical section, after a *release* is generated, it will mark this critical section as generating one *future miss* to all other processors. No mechanism prevents starvation, though the original program may do so.

4.2.2. Nested Locks

Some programs may acquire nested locks, i.e. they acquire a second lock when they already hold one. We do not model this situation and record it as sequence of two independent *acquires* and *releases*. At least for the benchmarks considered, this does not have a significant performance effect.

4.2.3. Counters

Updating a global counter appears as a successful *store-conditional* that is not followed by a *release*, since the critical section operation is actually performed by the *store-conditional*. In our statistical model, we identify *lock-numbers*, with no corresponding *release*, and mark them as counters. This informs the statistical simulator that no release instruction should be issued on this lock.

4.3. Barriers

There are two problems with synchronization barriers. First, they often occur at coarse granularity involving thousands or tens of thousands of instructions. For some benchmarks barriers are few in number, but cover a long time period. This is in contrast to ordinary instructions and other common statistical events that are large in number but require a brief period of time to complete. The second problem is that barriers typically involve all the processors, so they cannot be modeled in a statistically independent way in each processor. To deal with these problems, we scale down the proportion of execution between barriers in the detailed execution relative to the length of the statistical simulation. During the statistics-gathering phase, every time a processor is the first to reach a particular barrier, a new system-wide barrier number is assigned to it, so that all other processors know which barrier number they reached. We record the number of instructions each processor commits since it passed the previous barrier. At the end of the program execution the instruction counts are scaled to the total number of instructions each processor executed. The statistical simulator is then configured to generate a fixed number of committed statistical instructions, which it compares to the total number of instructions committed by the system during statistics gathering. The purpose is to scale down the amount of work done between barriers in proportion to the length of the statistical simulation.

Generating barriers has several disadvantages when it comes to simulation time. Since we in fact split the execution in multiple program phases, we should wait for each phase of statistical simulation to converge to a performance level before generating new barriers. However, it is necessary to know ahead of time the total number of committed synthetic instructions to execute in order to scale the barrier rates accordingly. To maintain accuracy, it then becomes necessary to overestimate the number of statistical instructions to generate and verify whether convergence has been reached during each phase. We used this technique to simulate barriers, and apart from a few benchmarks described in later sections, performance convergence was usually achieved within 1 million statistical instructions per processor.

5. Evaluation

In this section we compare statistical simulation of SMP systems with detailed, clock-level simulation using SimpleMP [8], a parallel, superscalar out-of-order simulator, which is based on SimpleScalar [9] core model.

5.1. System Model

The statistical simulator is used to simulate systems with 1 to 16 processors. The processors each use a statistical model of a superscalar processor as formulated in [4]

Table 2: Modeled memory architecture

Memory Architecture	Description
L1 cache	Instruction: 64K, 2-way Data: 64K 2-way 64 bytes cache lines
L2 cache	Unified 512K, 4-way associative 64 bytes cache lines
Shared Memory	1 to 4 banks, 68 cycles latency
Address Bus	Shared, FCFS 12 cycles latency before request sent to main memory
Data network	Shared, FCFS 16 bytes wide 20 cycles access latency(first chunk) 2 cycles for additional chunks

and summarized earlier. The memory architecture described in the previous section is implemented as a queuing system, with a queue for the shared bus, for the data bus, and for each memory bank. The statistical simulator uses the same parameter values used in Simple MP. The baseline parameters for the memory hierarchy configuration are summarized in Table 2. The baseline core processor parameters are given in Table 3.

5.2. Benchmarks

We consider main workload types: multiprogrammed workloads consisting of independent programs, and parallel, synchronized workloads. Multiprogrammed workloads are collections of SpecInt [10] benchmark programs. The performance of SpecInt benchmark programs is dictated by the CPU core performance, as well as the memory hierarchy performance. The performance of SpecFP benchmark programs are generally limited more

Table 3: Modeled processor architecture

Processor node	Description
Register Update Unit	64 entries
load / store Queue	32 entries
Fetch	Fetch 4 instructions per cycle 8 entry Instruction fetch Queue
Issue	4 instructions per cycle
Decode / Dispatch	4 instructions per cycle
Commit	4 instructions per cycle

by the memory hierarchy performance and are less interesting for study. The synchronized multiprocessor benchmarks from the Splash-2 benchmark [11] suite have inter-processor communication via shared variables and are synchronized via critical sections and barriers. Our benchmarks are all compiled using GCC with –O3 optimization. For initial statistics gathering, SimpleMP [8] was used and each benchmark was simulated with its reference input for 12 hours, which corresponds to approximately one billion total instructions.

5.3. Results

We evaluated statistical simulation on Splash-2 parallel benchmarks (for 1,2,4,8,12, and 16 processors), and a multiprogrammed workload of SpecInt95 (for 1,2,4, and 8 processors). Since detailed timing simulation of multiprocessor programs is very slow, it seemed practical to gather as many sets of statistics as possible during one single simulation run. However, this significantly increased the amount of memory required to execute the SimpleMP timing simulation. For multi-programmed workloads memory usage increases linearly with the number of processors, so we had to limit the study of multi-programmed workloads to 8 processors (1GB). Also, due to input requirements of some parallel benchmarks programs (again, because of SimpleMP detailed simulation; not the statistical simulator), some benchmarks could not be run for all configurations.

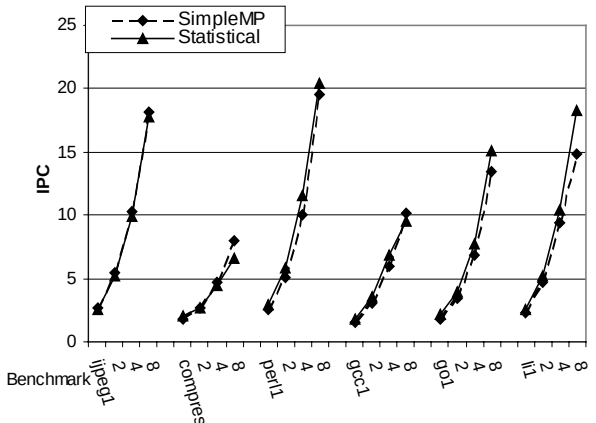

Figure 4: Real and estimated instruction throughputs for a simple Mix, on multi-programmed workloads, with 1 memory bank

5.3.1. Multiprogrammed Workloads

For reasons that are not inherent to statistical simulation, but rather due to our modified version of the detailed SimpleMP implementation, each workload consisted of multiple copies of the same SpecInt95 program. Each processor runs its own independent process, each having its own copy of the code segment and its own copy of the data segment. All memory references therefore access only private virtual pages. For these benchmarks, a single shared main memory bank is used. SpecInt benchmarks are targeted towards uniprocessors, and thus use the shared bus and shared memory less frequently than the parallel workloads.

Statistical simulation performance (total instructions per cycle or IPC) was measured at ten thousand cycle intervals. Performance was considered to have converged when the performance difference between two consecutive intervals was within 1%. Convergence occurred within 150K cycles for a single processor, and within 200K cycles for 8 processors. This translates into one and a half to two minutes on a 900Mhz Pentium when simulating an eight processor system. The corresponding detailed simulation takes 12 hours to complete -- a two order of magnitude speedup. Figure 4 shows performance results (measured as aggregate IPC) for the detailed SimpleMP simulation and for statistical simulation.

In general, the performance estimates of the statistical model for multiprogrammed workloads are very accurate (less than 10% error) for *Ijpeg*, *Perl* and *Gcc*, *Go* and *Li*, but it does show significant errors at 8 processors for *Compress (underestimated by 17%)*. On the benchmark reference input, *Compress* spends most of its execution time in one tight loop, which has been shown in [4] to be a cause of discrepancies between statistical and exact simulation for uniprocessors.

5.3.2. Scientific Parallel Workload Results

For parallel workloads, SPLASH-2 benchmarks were used. Initial simulations were performed with a single main memory bank system (which would create significant contention for larger systems), and follow up simulations use a four-bank system.

Figure 5 illustrates overall modeling accuracy as well as the importance of modeling cache coherence events when using one memory bank. The figure shows the baseline instruction throughput as given by detailed SimpleMP simulation; it also gives results with statistical simulation 1) assuming all memory requests require cache coherence actions (*all interventions*), 2) all requests are served by main memory (*No interventions*), and 3) cache coherence events are randomly distributed based on the program statistics (*statistical interventions*). We can see significant accuracy improvements with statistical cache event mod-

eling (up to 100% change in IPC for *Raytrace* or *Radiosity* with 16 processors). This workload characteristic is therefore important to model, and all following results statistically model cache interventions.

When interventions are statistically modeled, *Barnes-Hut* and *Ocean* accurately follow the exact performance curve when the number of processors is increased. *Raytrace* and *Radiosity* follow the trend accurately up to 12 processors, but the statistical simulation of *Fmm* significantly overestimates the performance speedup for configurations with four and more processors. The overall error across configurations is 29%, or 19% excluding *Fmm.* Statistical simulation converged within two to ten minutes for all programs and configurations, except *Radiosity* and *Raytrace* using 16 processors.

Because of the need to execute, on average, as many instructions inside statistical critical sections as the program does, we cannot scale down the length of critical sections to the total number of instructions necessary to reach convergence. Both benchmarks, *Radiosity* and *Raytrace*, make intensive use of lock synchronization, which considerably increases their convergence time to 25 and 45 minutes at 12 and 16 processors. This convergence time is independent of the number of instructions for which statistical data has initially been collected; it is only function of the contention for critical sections and their abundance.

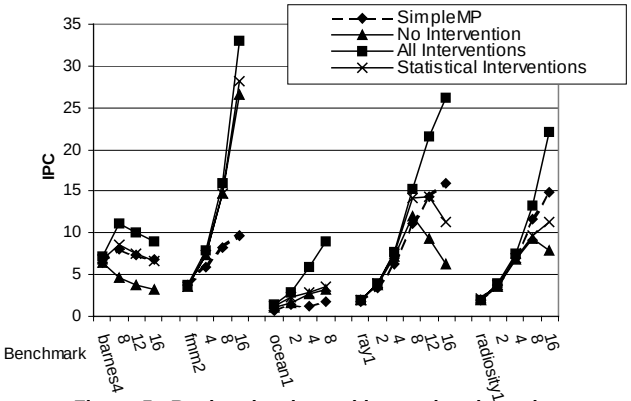

Figure 5: Real and estimated instruction throughputs with different approximations of cache interventions rates, for parallel workloads, on a system with 1 memory bank

Parallel programs share data most often by writing to shared pages. As explained earlier in Section 4, writing to a private page allows the processor to benefit from aggressive speculative execution of store instructions. To evaluate the impact of modeling this feature, we statistically simulated the workload assuming 1) all pages are shared (*all shared Vpages*), 2) all pages are private (*all private Vpages*), and 3) that their status is randomly distributed (*statistical Vpages sharing*) when the trace is created. As Figure 6 shows, the performance levels predicted by the statistical simulator do not change significantly. In the rest of this study, we randomly distribute the page sharing status, based on the distribution of consecutive accesses to a private page, as described in Section 4.1.

An important source of performance losses in parallel programs is synchronization and barriers. Results thus far have included *acquires* and *releases* as part of the basic mix; we now consider the importance of barriers, which are more difficult to model due to the granularity issues described earlier. Figure 7 compares the performance predicted by detailed SimpleMP simulations, with performance predicted by statistical simulation using 1) the basic instruction distribution, but with no barrier instructions (*Statistical*), and 2) a statistical simulation with randomly generated barrier instructions in the program trace (*Statistical+Barriers*), according to the technique described in Section 4.3. For these simulations *Fmm* is the only benchmark to show improved accuracy from adding statistical barriers, reducing the error to 32% from 150% with 16 processors. Other benchmarks are not affected

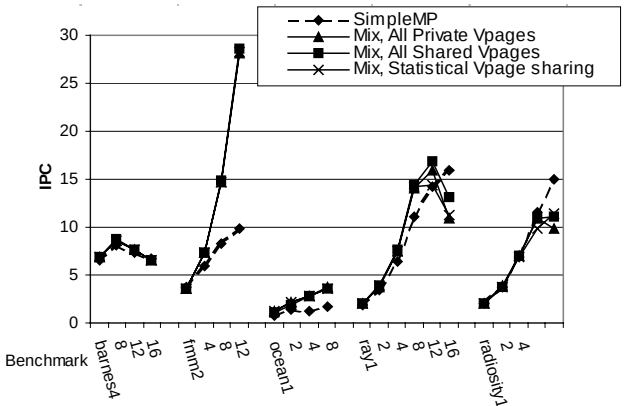

Figure 6: Real and estimated instruction throughputs with different approximations of access rates to shared and private pages, for parallel workloads, on a system with 1 memory bank

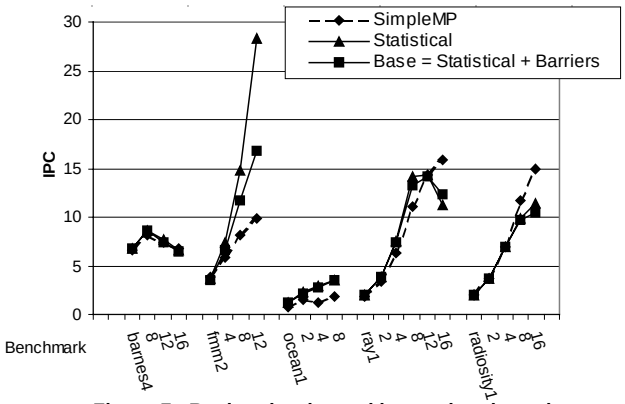

Figure 7: Real and estimated instruction throughputs with and without generating statistical barriers, for parallel workloads, an a system with 1 memory bank

because the barriers issued by their processors are tightly grouped in time; i.e. they tend to reach the barrier at about the same time. This improvement brings the average error for configurations with 12 processors or less to 15%.

Moreover, the statistical performance trends of all benchmark programs precisely follows the performance trends of execution-driven simulations. The performance of *Barnes-Hut* degrades by 18% between 8 and 16 processors, while statistical simulation suggests a performance degradation of 24% between the same configurations (Figure 7). The statistical performance curve of *Raytrace*, accurately follows the performance curve given by SimpleMP up to 12 processors. It shows between 12 and 16 processors a performance loss (-20%) with statistical simulation while SimpleMP shows a low performance increase (+9%). The statistical and execution-driven performance curves of *Fmm* and *Ocean* are comparable up to 8 processors after which they start diverging. Additional results show that this discrepancy can be partly reduced by using more complex instruction profiles, as shown in [4].

5.3.3. Quad-Memory Bank Systems

The preceding set of experiments used one memory bank, which puts high pressure on the shared memory. We relax this condition here and study which statistical models are most accurate for 4 memory banks. Results are given in Figures 8 and 9.

For this system configuration, the lower memory utilization does not require the addition of cache interventions to improve accuracy significantly (Figure 8). However, as with a single-bank system, the widely dispersed barriers in *Fmm* require the modeling of barrier generation to achieve good accuracies (Figure 9). Overall, statistical modeling of barriers, synchronization, and the simple *mix* give accurate performance estimates for *Barnes-Hut* (11% average error), *Raytrace* (9% average error) and *Radiosity* (up to 8 processors with 4% error). However it shows less accuracy on *Fmm* (25% average error) and *Ocean* (70% average error).

This simple model is sufficient to identify the number of processors after which there is a diminishing performance return. For eight and more processors, execution-driven simulation of *Barnes-Hut* shows a strong performance degradation (19%) from 8 to 16 processors (time-constrained simulations), which correlates very well with the statistical simulation results (15% performance loss). Likewise, statistical simulation clearly identifies the point of diminishing performance return for *Ocean* after eight processors, and *Raytrace* after twelve processors. For *Radiosity* and *Fmm*, which have more consistent speedups with respect to the number of processors, statistical simulation results accurately follow their performance trend up to 8 processors. We did perform additional simulations with *Ocean* and *Fmm* using more detailed instruction mix modeling as described in [4]; these more accurate instruction mixes can reduce the error in *Ocean* by half, and *Fmm* by 15%.

6. Conclusions

Recent proposals investigated the applicability of statistical simulation for microprocessor design. Program statistics are collected during an initial execution-driven simulation of a benchmark, and later used to predict performance trends for a wide variety of architecture changes, one to two orders of magnitude faster than conventional simulation. In this paper, we extended the method to symmetric multiprocessor systems. We first described the set of statistics that enable statistical simulation of multiprogrammed multiprocessor systems, running workloads made of SpecInt95 benchmark programs. Their statistical simulation converged within one to two minutes with a small 10% average error, for configurations of 1 to 8 processors, and can be run over a wide range of architecture parameters.

In order to accurately simulate synchronized parallel

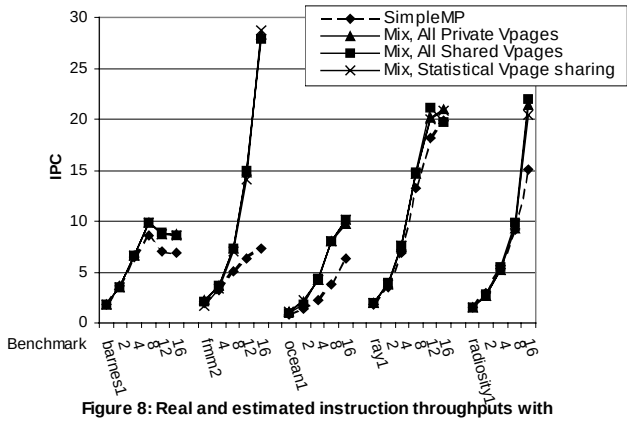

Figure 8: Real and estimated instruction throughputs with different approximations of access rates to shared and private virtual pages - for parallel workloads, on a system with 4 memory banks.

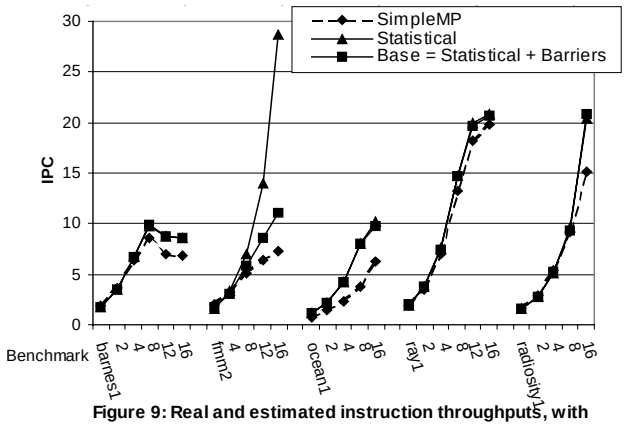

Figure 9: Real and estimated instruction throughputs, with and without generating statistical barriers - for parallel workloads, on a system with 4 memory banks

scientific workloads, statistics on cache interventions, barriers distributions, lock accesses, and critical section mixes are added to the uniprocessor statistics. Our results show that it is possible to achieve accurate performance estimates on parallel programs from the Splash-2 benchmark suite within two to ten minutes of simulation. On a single-bank system statistical simulation of parallel benchmarks leads to 15% average error for configurations with 1 to 12 processors, and 25% average error for all configuration with 16 processors but Ocean. On a quad-bank system, statistical simulation leads to 12% average error on configurations with 1 to 16 processors running Splash-2 benchmarks programs, apart from Ocean. Statistical simulation also very accurately predicts the speedup (i.e. the trend) for Ocean with increasing number of processors, but has a higher absolute error (up to 85%) on individual configurations.

Perhaps more important than the absolute errors is the ability to predict performance trends as shown in the performance curves. The trend predictions are sufficient to get initial performance estimates in a few minutes in most cases. For the cases with larger errors, using more detailed modeling of instruction distributions does sometimes help (at the cost of longer convergence time) and almost halves the error on Ocean.

Overall, we conclude that statistical simulation for SMP systems is a useful tool for investigating design options quickly. The simulation technique usually converges to a performance estimate within minutes, can be easily used to evaluate a wide range of system design tradeoffs.

Acknowledgements

This work was supported in part by National Science Foundation grant CCR-9900610, by IBM Corporation, and Intel Corporation. The authors would also like to thank Ho-Seop Kim for his help with early versions of the statistical simulator and Timothy Heil for his valuable comments on this work.

References

[1] R. Carl and J. E. Smith, “Modeling Superscalar Processors via Statistical Simulation,” *Workshop on Performance Analysis and Its Impact on Design*,” June 1998.

[2] L. Eeckhout, K. DeBousschere, and H. Neefs, “Performance Analysis Through Synthetic Trace Generation,” *Inernational Symposium on Performance Analysis of Systems and Software*

[3] M. Oskin, F. T. Chong, and M. Farrens, “HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Design,” *Proc. 27th International Symposium on Computer Architecture*, June 2000.

[4] S. Nussbaum and J. E. Smith, “Modeling Superscalar Processors via Statistical Simulation”, May 2001

[5] P. Sweazy and A. J. Smith. “*A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus”*, In the 13th International Symposium on Computer Architecture, 1986, pp. 414- 423.

[6] Chris Gniady, Babak Falsafi, and T.N. Vijaykumar “ Is SC + ILP = RC ?“, In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[7] C. Scheurich and M. Dubois, "*The Design of a Lockup-Free Cache for High-Performance Multiprocessors*," Proceedings of Supercomputing '88, pp. 352-359, 1988

[8] Ravi Rajwar, Personal communication, SimpleMP, multiprocessor simulator developed as part of the Galileo project. http://www.cs.wisc.edu/~galileo

[9] Doug Burger and Todd Austin, “The SimpleScalar Tool Set, Version 2.0, *Computer Architecture News*, pp. 13-25, June 1997.

[10] “Standard Performance Evaluation Corporation (SPEC2000 *CPU benchmark*)". http://www.spec.org/osg/cpu2000/.

[11] Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., and Gupta, A.,"*The SPLASH-2 Programs: Characterization and Methodological Considerations*", *Proc. 25th International Symposium on Computer Architecture (ISCA '98)*, Barcelona, July 1998, pp. 180-191.

[12] P. Bose and T. M. Conte, “Performance Analysis and Its Impact on Design,” *IEEE Computer*, pp. 41-49, May 1998.

[13] Rajagopalan Desikan, Doug Burger, and Stephen W Keckler, “Measuring Experimental Error in Microprocessor Simulation”, In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, July 2001.

[14] Y. Yan, X. Zhang and Z. Zhang, "Cacheminer: a runtime approach to exploit cache locality on SMP", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 11, No. 4, 2000, pp. 357-374.