Execution-Driven Simulation of Multiprocessors: Address and Timing Analysis

S. Dwarkadas

Department of Computer Science

J. R. Jump and J. B. Sinclair

Department of Electrical and Computer Engineering

Rice University, Houston, TX77251-1892 [1]

Abstract

This paper describes and evaluates an efficient execution-driven technique for the simulation of multiprocessors that includes the simulation of system memory and is driven by real program workloads. The technique produces correctly interleaved address traces at run time without disk access overhead or hardware support, allowing accurate simulation of the effects of a variety of architectural alternatives on programs. We have implemented a simulator based on this technique that offers substantial advantages in terms of reduced time and space overheads when compared to instruction-driven or trace-driven simulation techniques, without significant loss of accuracy. The paper presents the results of several validation experiments used to quantify the accuracy and efficiency of the simulator for sequential, distributed, and shared-memory multiprocessors, and several parallel programs. These experiments show that prediction errors of less than 5% compared to actual execution times and overheads 6 to 30 times lower than those incurred by cycle-level simulation can be achieved. Predictions of relative performance metrics such as speedup tend to be even more accurate, making this technique especially attractive as an efficient method for comparative investigations of parallel system designs.

Categories and Subject Descriptors: B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids (uniprocessors and multiprocessors) — execution-driven cache simulation, dynamic tracing; C.4 [**Computer Systems Organization**]: Performance of Systems — measurement and modeling techniques;

General Terms: measurement, performance, verification

Additional Key Words and Phrases: execution-driven simulation, parallel tracing, shared-memory multiprocessors, distributed systems

1 Introduction

Parallel computers are playing an increasingly important role in providing the computing power necessary for solving large, computationally-intense problems, and this trend is likely

[1]This research was funded in part by the Center for Research on Parallel Computation through NSF Cooperative Agreement No. CCR-9120008 and by NSF and ONR under ONR Grant No. N00014-87-K-0324.

to accelerate. As a result, we see a growing need for efficient techniques to predict the performance of parallel computer systems, especially with regard to larger machines. It is important to evaluate proposed architectural features before incurring the cost of building such large-scale multiprocessors. Evaluation tools are useful for the performance debugging of existing as well as proposed systems.

Analytical models and distribution-driven simulation may provide relatively quick estimates of performance but often lack sufficient accuracy because of the difficulty in characterizing the behavior of real programs stochastically. Simulation of the execution of real programs has proven to be a more effective tool in designing new systems and in analyzing existing ones. Instruction-level emulation of program execution is a highly accurate simulation methodology but involves large overheads. Modeling shared memory and/or cache memories magnifies these overheads. Trace-driven approaches typically require large amounts of storage in addition to the time overhead. Also, generating accurately interleaved multiprocessor traces is difficult, since changes in this interleaving due to timing variations or changes in the addresses themselves due to timing-dependent program behavior will not be reflected in the traces.

We have developed and evaluated an efficient technique for the simulation of multiprocessors that includes the simulation of system memory and/or caches and is driven by real programs. The method used is a variant of *execution-driven simulation* [8, 7]. Our goal is to speed up the simulation without a significant loss of accuracy compared to other techniques such as instruction-driven simulation, and without incurring the large storage overheads of trace-driven approaches or requiring special hardware support. Simulation of the architecture during the actual execution of the parallel program under consideration permits us to determine the effects of the architecture on the execution of the program with a high degree of accuracy. We have implemented this technique as part of the Rice Parallel Processing Testbed [8], a simulation tool for the performance evaluation of parallel computer systems.

In the RPPT, we perform a compile time analysis of the program and convert it into one that will provide information on instruction execution times and memory accesses to the simulator as it executes. This information can then be used to drive a simulation of the architecture at various levels of detail, depending on the accuracy required and the architectural features under study. The program modifications are non-intrusive; they do not affect the logical execution of the program nor its simulated timing behavior. The address trace produced during the simulation by each thread or process is identical to the sequence of addresses produced by the unmodified program.

In this paper, we provide extensive validation results. The determination of accuracy is based on validation studies in which we compared the timing estimates generated by the simulator with measurements made on sequential and shared-memory multiprocessors, as well as with predictions produced by a cycle-level simulator of a proposed parallel system. Our results show that we have developed an efficient simulator with an accuracy of within 5% for the Sequent Symmetry[2] [20]. The errors are mostly due to the variable execution times of the complex instructions, which our simulator does not attempt to predict. The simulation time observed is about 600 times as great on average as the actual uniprocessor execution time of the program for a detailed simulation (including simulation of the network and set-associative cache) of every instruction and data address referenced. The overhead is highly dependent on the degree of detail of the architecture and the number of process interaction points simulated. For the simulation of private caches on a distributed memory machine, the overhead is between 60 and 180 for a set-associative cache.

In the next section, we provide an overview of the execution-driven approach, validate the profiling technique, and explain the details of our simulator. Section 3 provides an evaluation of our simulation approach on the Sequent Symmetry [20] as well as a comparison with a cycle-level simulator [16]. Section 5 concludes by providing an overview of our work.

2 The Execution-Driven Technique

In execution-driven simulation, the execution of the program and the simulation of the architecture are interleaved. The execution of a parallel program on a target parallel architecture is simulated by the pseudo-concurrent execution of the same program on a uniprocessor host. Unlike instruction-level simulation where the execution of each machine language instruction is emulated by the execution of many host instructions (even when the host and target machines are nearly identical), execution-driven simulation executes the machine language instructions of the program directly on the simulation host. This avoids the time overhead involved in emulation, while retaining most of the accuracy of instruction-level simulation. During execution-driven simulation, the simulation time counter needs to be updated only at the beginning of a basic block or at every point where two or more processes can interact. A basic block is a maximal contiguous set of instructions, all of which will be executed exactly once if the first is executed, and conversely [30]. Thus, any instruction that is the target of a branch and any instruction following a branch starts a basic block. Event list

[2]Use of the Sequent Symmetry S81 was provided by the Department of Computer Science at Rice University under NSF CISE Infrastructure (II) Grant CDA-8619393.

manipulation overhead is incurred only at process interaction points. Such interactions can include message passing, synchronization, cache misses, and accesses to shared variables.

Our technique generates timing information and/or address traces for the program *on the fly* [11] during execution. This avoids the large space overhead incurred by trace-driven simulation and the time involved in accessing these large traces from disk, as well as the specialized hardware needed to generate these traces [1, 6]. The execution-driven technique achieves its goal of avoiding the high overhead associated with instruction-level simulation while retaining most of its accuracy by extracting as much information as possible from the program at compile time. Execution-driven simulation also efficiently tracks dynamic variations in the address traces and their interleaving due to architectural changes or timing dependent program behavior. Figure 1 shows the basic cycle of the execution-driven simulator.

The profiler, which is an important component for simulating shared memory systems, is described briefly in the next subsection. A detailed analysis, description, and validation of the profiling technique can be found in [12] and [14]. The description of the profiler is followed by an overview of the simulation system that we have developed.

2.1 The Address Trace and Timing Profilers

Conventional simulation techniques used to predict the performance of system memory involve two main steps. First, detailed address traces must be generated to characterize the workload. Second, these traces must be applied to a simulation model of the memory system. The execution-driven approach to trace generation and simulation eliminates most of the overhead associated with conventional trace generation by using an on-the-fly technique.

A preprocessor or *profiler* parses the program's assembly code at compile time. Profiling or preprocessing is done at the assembly language level rather than the machine code or high level language level, since the assembly code is easily modified in order to capture the run time information required. The assembly code also most closely represents the instructions executed on the target machine. The information extracted is used to modify the assembly code so that timing and address trace information is generated during program execution (hence the term execution-driven simulation).

The profiler parses the program's assembly code to extract the following information:

1. address references that can be used to simulate the underlying memory architecture and cache organization.

2. timing information that is used to update simulation time.

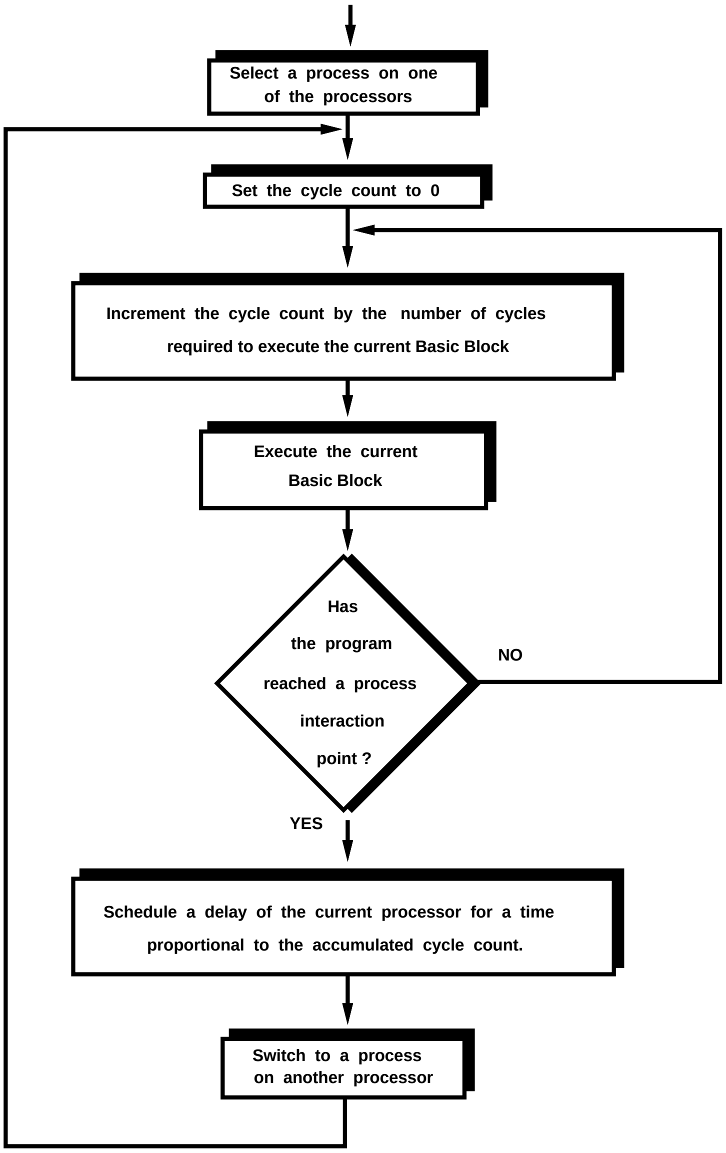

Figure 1 The Basic Execution-Driven Simulation Cycle

3. the type of memory access (for example, read or write, or instruction or data).

Overhead at compile time is a constant one time overhead. Ideally, all the address trace and timing information for a basic block would be available at compile time. If this were true, it would allow us to simulate the effects of the entire basic block's execution at the start of each basic block, resulting in a very efficient simulation.

While compile time extraction of all timing information and instruction addresses is possible, not all data addresses can be determined at compile time. Hence, some dynamic address extraction is necessary. The effective address of an indexed or indirect data reference (one whose address is contained in a register, for example) is an example of an address that cannot be determined at compile time. In order to determine such an address, code must be inserted in the program's assembly code before every such reference to extract the address at run time. We do this for all data accesses to ensure that all addresses generated are for the same virtual mapping. Hence, while execution-driven simulation is very effective for systems that do not require address tracing for evaluation, it is less effective for an evaluation of systems that requires address tracing.

Figure 2 summarizes the manner in which the dynamic address trace is generated and processed. The profiler creates an address template for each basic block, which contains the address and timing information for all instruction references, and markers to indicate the presence of data references that will be determined at run time. A template is needed per basic block in order to perform as much work as possible at compile time and to ensure the accuracy of the resulting address trace since by this manner only the effect of instructions actually executed will be simulated. At the start of each basic block, a subroutine that processes the address template for the basic block (generated at compile time) is called. This routine reads in addresses from the template until a marker indicating that a data address is accessed by the program at that point is reached. It then returns and the basic block continues execution until the instruction that accesses the data address is encountered. The instructions inserted to compute the data address are then executed, along with the routine that continues reading addresses from the address template until the next marker or the end of the basic block is reached. The net effect is to supply the cache simulator with an accurate dynamic address trace in which instruction and data addresses are interleaved in the proper sequence and at the exact times at which they would be accessed during actual unprofiled execution. The addresses can be processed dynamically by the cache simulator, or they may be stored for later analysis.

Figure 3 provides an overview of the steps necessary to produce a profiled executable. The source program is compiled to produce an assembly language version as well as an executable (if the program is being profiled to generate instruction addresses). These are then fed to the profiler, which uses information from the executable's symbol table (to generate the instruction addresses and address template, if desired) to produce a modified assembly language listing of the original program that provides the necessary tracing information at run time. This modified program may then be assembled and linked with the subroutines that perform the collection of run time information and the cache simulation. The result is a profiled executable image that can be used to determine the performance of the original program.

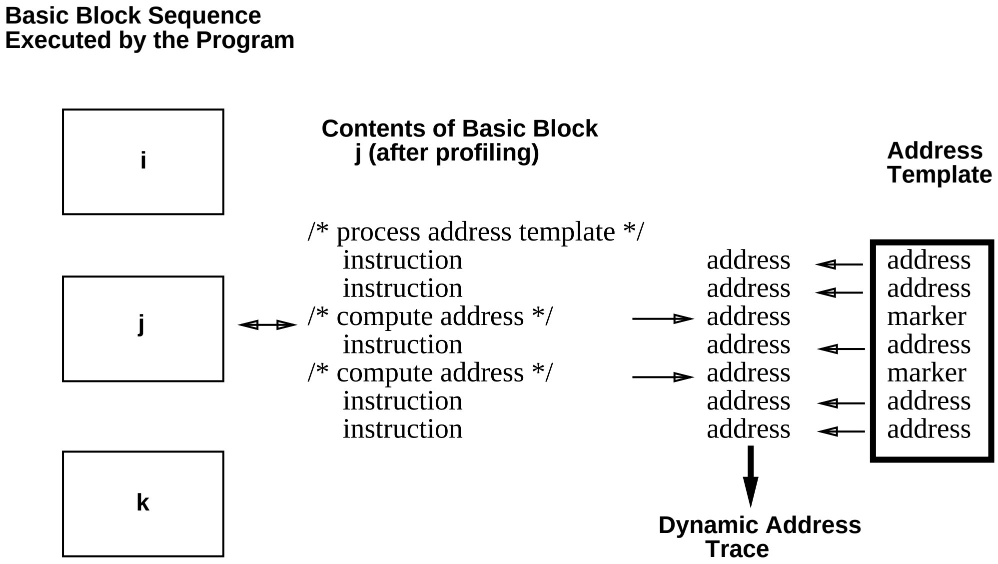

Figure 2 Dynamic Address Tracing for Cache Simulation

The above process results in profiled code that generates a detailed execution trace of the program as it executes. The user has the option to either generate all instruction and data addresses accessed by the program, generate no address trace and use only timing information, or generate only data addresses and ignore the simulation of any instruction addresses. If only data addresses are required, the address template per basic block need not be created or examined.

The execution-driven methodology has been used to implement profilers for the Motorola 68020, Intel 80386, and SPARC processors. The 68020 profiler was validated in [14]. The ratio of simulation time to actual workload execution time is called the slowdown factor or overhead. We achieve an overhead of 15 to 48 for address trace generation alone for the 68020 processor (this includes a call to the simulation routine to reconstruct the address trace from both compile time and run time information and the generation of a trace of all addresses including the instruction addresses, but does not simulate the effects on the cache of the addresses). Since the address trace is processed as it is generated (no file I/O involved), the overhead in cache simulation is also reduced, resulting in a total overhead of 35 to 90 for the simulation of a direct-mapped cache. The overhead associated with address trace generation using instruction-level simulation is reported to be on the order of 1000 [1]. Using a hardware tracing facility (as in the T-bit of the VAX processor) that interpretively determines instruction and data addresses, the overhead is about 100. If microcoding is used, the overhead is about 10. These figures do not include cache simulation. By extending

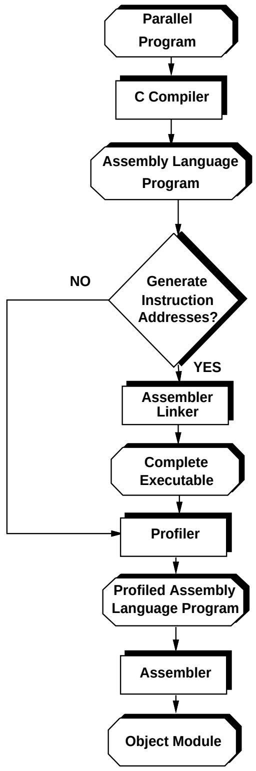

Figure 3 Profiling Steps

the execution-driven concept to the dynamic generation of address traces and the simulation of caches, we have developed a space efficient approach that is significantly faster than conventional instruction-level simulations. In the next section, we present a summary of our uniprocessor validation results (see [14] for a more detailed analysis).

2.1.1 Profiler Validation

The simulation system was validated to determine the accuracy of the execution time predictions made by the simulator. The timing validation was performed by comparing measured times for the execution of an unprofiled program on the SUN–3/280 with the times generated by the program's simulation.

The system for which the timing validation was carried out was a SUN–3/280. It has a MC68020 processor with a 25 MHz clock, a MC68881 floating point coprocessor with a 20 MHz clock, and a 64 Kbyte direct-mapped write-back cache with a line size of 16 bytes. The access times for main memory are 360 nsecs for a read and 680 nsecs for a write. The MC68020 is a pipelined processor with an on-chip instruction cache. The hit rate of the on-chip cache was only estimated to be ∼60% (an average for most applications) and not actually simulated, in order to reduce simulation complexity. The timing measurement for the unprofiled program was performed with a hardware clock and software that ensures that only user time for the process is measured.

Figures 4 and 5 present the percentage error in simulation time predicted for the FFT and Mergesort (MERGE) programs as the data set size and therefore the running time is increased (1024 to 16384 points for FFT, 4096 to 65536 integers for MERGE, with time taken varying between 0.5 and 15 seconds). The error is defined as the ratio of the difference between the execution time obtained from simulation and the real execution time of the program, compared to the real execution time of the program. The FFT program is floating-point intensive, while the Mergesort program has no floating point computation. The error is a maximum of -4% for the FFT program and a maximum of +4% for the Mergesort program. Since the error is almost constant, it can be attributed to pipelining and instruction cache effects that are not being modeled. The positive error shown by the Mergesort cache simulations may be due to a larger actual hit ratio than that assumed for the on-chip instruction cache (the Mergesort program has a short inner loop in the merging step). The accuracy was also limited due to the 20 msec clock resolution of the Sun–3/280, and to instructions whose execution times depend on the values of the operands.

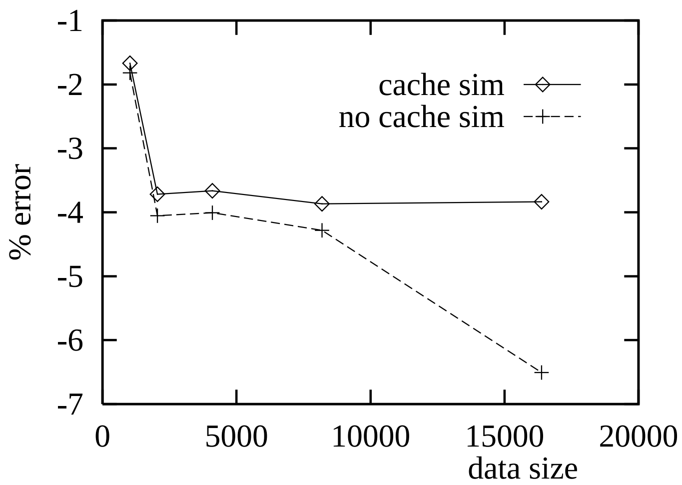

Figure 4 Percentage Error in Uniprocessor Simulation - FFT, 68020 processor

The error in prediction by the profiler without simulation of the cache, that is, assuming that all cache accesses are hits (*no cache sim*), has also been plotted on the same graphs for comparison. The error increases as the data size used by the program increases (caused

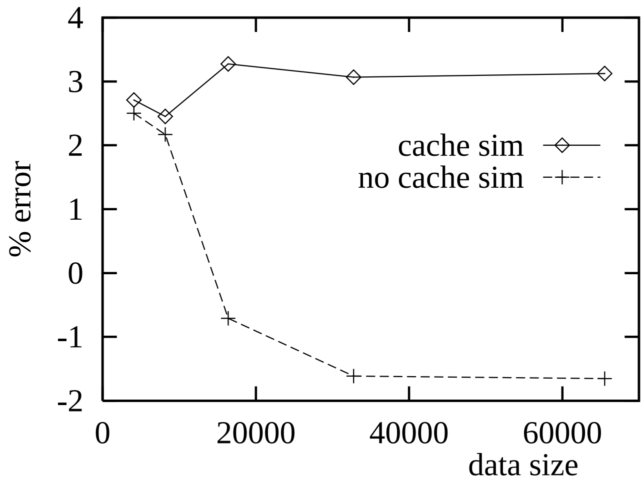

Figure 5 Percentage Error in Uniprocessor Simulation - MERGE, 68020 processor

by an increase in the miss ratio), while the error with simulation of the cache stays fairly constant, indicating a good prediction of the cache hit ratio.

2.2 The Rice Parallel Processing Testbed

The Rice Parallel Processing Testbed (RPPT) is a performance evaluation tool for studying the execution of parallel programs on concurrent systems. It uses execution-driven simulation to accurately describe the data dependent flow of execution in a parallel program and to efficiently provide estimates of user code execution times. Detailed simulation models of the interconnection structure of the system being simulated account for delays in performing interprocessor communication, remote process synchronizations, or remote data accesses. During the execution, the timing information from the architecture model is combined with the estimates of instruction execution times to obtain accurate estimates of the overall execution times of a parallel program on a chosen architecture.

The parallel programs under evaluation may be written using Concurrent C [21] or Presto [2]. The shared memory parallel programming model presented to the user consists of lightweight threads executing in a single address space. These threads synchronize using flags, locks, barriers or semaphores (Concurrent C), and monitors or condition variables (Presto). Programs may also use the PARMACS parallel programming macros [4].

The basic components of the RPPT software system are as follows:

- Concurrent C: This is the C programming model provided to the user [21]. It provides the user with primitives to design both shared and distributed memory programs.

- Presto : This is the C++ programming model provided to the user. Presto, written in C++, is an object-oriented programming environment designed for shared memory systems and developed at the University of Washington, Seattle [2]. A Presto interface has been provided to the user by porting Presto on top of Concurrent C [24].

- YACSIM (Yet Another C SIMulation Package): This is a process-level, discrete-event simulator providing a set of procedures for process creation, delaying, event queue manipulation, and statistics collection [9, 18]. YACSIM primitives may be used to model any architecture being simulated.

- Address and Timing Profilers: A program analyzer is used to modify the assembly language version of the program so that address and timing information may be extracted at run time. These modifications include the appropriate calls to the architecture routines.

Figure 6 shows the simulation methodology used in the RPPT. The parallel program is written in Presto or Concurrent C and is profiled with one of the address and timing profilers. The profiler inserts code which is used to generate timing and/or address information and to make calls to the architecture routines that process this information. The profiled code is then linked with the simulation libraries to create the final simulation executable. The resulting executable is one that provides accurate estimates of performance, allowing the user to add various performance probes without perturbing program and system behavior.

The simulation of a parallel processor on a sequential processor entails the serialization of events on the simulated machine. While this does not affect the accuracy of the predictions (the interleaving of events is accurately maintained), it does affect the simulation time. The use of light-weight threads for both user processes and architecture model initiated events also implies that enough memory (both physical and virtual) is needed to hold all the program text and data that would be used on the parallel processor. Each thread or process is allocated separate stack space on the heap. This may limit the size of a simulation experiment even in a virtual memory environment because of constraints on swap space.

Every simulation may be configured to simulate the architecture and program at different levels of detail either at compile time or at run time. In the case of message-based systems, the program may be profiled to generate only timing information at the beginning of each basic block. The message transmissions themselves are simulated in great detail. This results

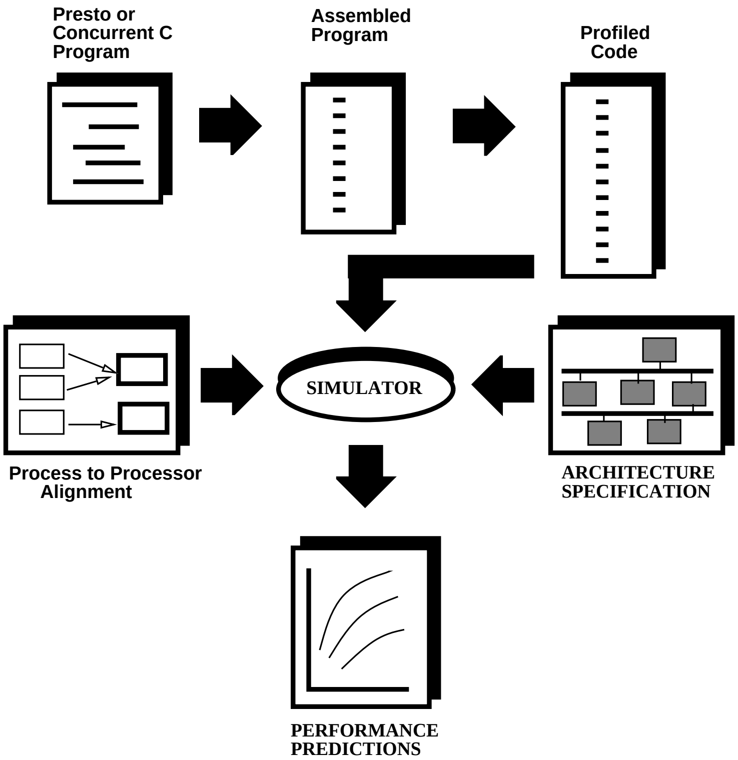

Figure 6 The Rice Parallel Processing Testbed

in program slowdowns of 1.2 to 15 [7] at 16 processors with slowdown factors depending on the frequency of communication of a particular application. If the performance of the system caches must be determined, address trace information may also be generated. In this case, the cache data structures and timing information must be updated, but there is no coherency maintainance overhead for a message-passing system of this nature. The overhead involved is 60-180 [14] for the simulation of a set-associative cache with a write back policy, as compared to the Concurrent C version of the program. The program is rescheduled only at process interaction points, which may be only the message sends and receives. Figure 7 presents the overheads for two-processor simulations of two programs, Eigenvalue/Eigenvector Decomposition (EIGEN) and Mergesort (MERGE), with different architectural features simulated in detail. The two-processor simulations represent worst-case behavior in terms of simulation time for most programs and architectures, since the overhead when compared to real time decreases on average as the number of processors is increased. The results show the sensitivity of the overhead both to the level of cache simulation detail (e.g., direct-mapped (`dmap`) vs. set-associative (`setass`) cache, network contention (`bus` vs. `nobus`) simulation) as well as to the number of context switches involved (write-through-write-allocate (`wtwa`) vs. write-back (`wb`) policy when simulating bus activity). The overhead added by the trace generation code alone is shown by `tracegen`.

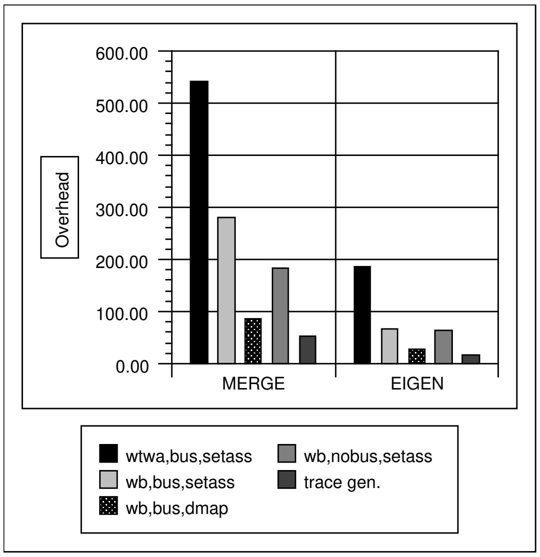

Figure 7 Overhead Comparisons for Different Levels of Multiprocessor Simulation Detail (2 processors)

In the case of shared memory multiprocessors, once again varying degrees of detail may be simulated. The architecture may or may not have caches. In certain studies, it may be adequate to simulate only synchronization activity, and hence these may be the only accesses simulated in detail. One could also examine only shared data accesses, all data accesses whether private or shared, or instruction and data accesses. When distinguishing between private and shared data, some run time overhead is necessary.

The simulator approximates the effects of operating system activity using estimates. If the operating system code were incorporated as part of the user code or as part of the architecture model, its effects could be accurately determined. The effects of I/O are ignored in the simulations carried out here. Also the effects on performance of virtual to physical memory mapping and translation are ignored. Detailed models could be provided for such simulation. Analytical models may be used at any level to replace the detailed simulation of a system component.

2.3 Simulation of Shared Memory Multiprocessors

The simulation of shared memory systems requires knowledge of the addresses accessed by the program in addition to the timing information in order to model the complex interactions between the different processes. The presence of caches in most shared memory systems results in an increased architectural as well as simulation complexity. Any point in the execution of a process where its actions may affect or be affected by another process must be treated as an event in the event-driven simulation for accurate performance prediction (an interaction point).

The shared memory application is profiled as described in the previous section to extract address and timing information. The architecture simulator determines the effect of each address access on the current process's processor cache (and the rest of the system if applicable), and accumulates simulation time for the process using the information supplied by the profiling code. All registers used by a process must be saved and restored every time the process is switched out in order to simulate any other process. Whenever an interaction point is reached, the current process is delayed for an amount of (simulation) time equal to the time accumulated by the actual execution of the code. Interaction points can include synchronization, cache misses, and accesses to shared variables, depending on the level of detail of the simulation.

In the simulation of shared memory programs, we have no explicit method of determining when a shared variable is referenced within the program. Although the shared variables used in the program are declared explicitly, we still cannot determine at compile time whether a shared location is accessed. This is because global addresses may be assigned to local pointer variables within the program, and these assignments do not indicate at compile time whether the address that is finally generated is the address of a local or shared (global) variable. Hence, the type of access must be determined at run time after the address has been supplied to the architecture simulator by the profiling code.

The partitioning of the virtual address space is achieved by means of the address mapping primitives provided by Concurrent C. These primitives manipulate a data base of address ranges and their associated type codes. They are used to determine the manner in which an address reference is treated. They may be used to model the partitioning of data among the processors in a multiprocessor system, or to determine the coherency protocol to be used in dealing with different data types. Address ranges are entered into the data base and associated with a unique type code before the program begins execution. Address ranges are assumed to be non-overlapping. Once we ascertain the manner in which an address reference must be treated (location/type), the overhead incurred in making the access can be determined, and the corresponding access can be simulated by the architecture model of the target machine.

There are two distinct components to a typical shared memory architecture - the cache and memory model, and the network model. Each component is modeled with the help of the semaphore, queue, process and state variable primitives provided by YACSIM. For example, the bus in a shared bus architecture may be modeled as a blocking queue, where the processor performing the bus access is blocked so that no other process can run on it. Each processor cache may be modeled with a semaphore to protect access to the cache resource, as well as a cache data structure that keeps track of the tag and state information for the cache. The two components to the architecture model are provided with a well-defined interface, and interact closely with one another.

3 Evaluation

We have presented validation data for message-based architectures in [7]. In this section, we present validation results for shared memory architectures (some data is also presented in [13]). In particular, we compare results produced by a simulation of the Sequent Symmetry against measurements on the corresponding real machine. We also compare the execution-driven simulation of a SPARC-based multiprocessor with a corresponding instruction-level simulation of the same system in order to evaluate the accuracy and efficiency of execution-driven simulation relative to this more conventional approach to simulating computer systems.

3.1 The Sequent Symmetry

The Sequent Symmetry [27] is a single bus shared memory multiprocessor with a private cache at each processor. The coherence protocol is write back with invalidation of other copies when the data is shared, similar to the Berkeley protocol [19]. In addition, when a cache that has the dirty copy of a cache line supplies the data, the cache also invalidates its own copy regardless of whether the request was for a read or a write access. Coherence is maintained by using the bus network's snooping ability. The bus uses split transactions.

In order to determine the effects of the cache organization on performance, all accesses generated by the program must be simulated. If the reference is to a location that has been identified as shared, coherence must be maintained as well. The simulation overhead of maintaining coherence is avoided for private data (such as instruction and stack space) that is not shared between processes. In addition, event-list manipulation may not be required when private data hits in the cache, since actions on private data are not affected by the execution of any other process.

A data structure is maintained for each cache and the effect of each address reference on the cache is simulated. Execution time for the processor is accumulated without incurring any event list manipulation overhead until a point is reached where the process may interact with another process either directly or indirectly. This reduces simulation overhead to the minimum possible, since it is necessary to reschedule and delay a process only if its activity or the activity of some other process may affect its own or some other process's execution path. A cache miss constitutes an interaction point since a shared resource must be accessed and the effects of the miss simulated. A shared memory access also constitutes an interaction point, since coherence must be maintained.

Accesses to synchronization data (locks) must be treated specially since they imply atomicity. This involves a modification of the synchronization routines provided by the programming model to simulate the necessary resource contention. In the case of the Sequent Symmetry, synchronization is achieved by using the coherence mechanism to bring in an exclusive copy of the cache line in which the lock resides. The synchronization operation is then performed atomically in the cache.

We modeled the bus network as a round robin queue (a resource that is provided by YACSIM). Access to each cache resource was controlled by a semaphore, to simulate contention for the resource. Memory contention was also modeled.

3.1.1 Validation

The programs used to validate our simulator were parallel versions of the Subgraph Isomorphism, Fast Fourier Transform, Gaussian Elimination, and Mergesort programs. These programs were chosen to represent a spectrum of numeric and non-numeric scientific programs. The simulation was validated by timing the execution of these programs on the Sequent using its microsecond clock timer, and then comparing these times with those predicted by the simulator.

The Sequent on which the validation was done is a 20 processor Symmetry with 80386 processor nodes that have 64 Kbytes of cache (2 way set-associative, 16 byte lines) associated with them. The cache line replacement policy implemented by the Sequent architecture is the least recently used policy. The processor speed is 16 MHz and the bus speed is 10 MHz.

Figures 8 and 9 compare the speedups predicted by the simulator (*sim*) with the actual speedups obtained (*real*) for the Gaussian Elimination program (GAUSS) operating on a 96x96 matrix. The problem was divided among the processors by row, with each of the n processors working on every *nth* row. At every iteration, the computation time between synchronizations decreases since there is one less row to work on. However, since the work is evenly distributed with limited serialization, good speedups are achieved. As can be seen, both the time and speedup prediction errors are fairly constant, indicating that the simulator modeled the sequent architecture well. The timing errors (which are comparable to the error in simulation of single 80386-based processors) may be attributed to inaccuracies in real time measurement, as well as to the data dependent execution times of some instructions, which cannot be predicted at compile time. If the speedup predictions are compared to the speedups actually realized, the average error in prediction is ~0.25%, which is much lower than the 3.8% average error in execution time prediction.

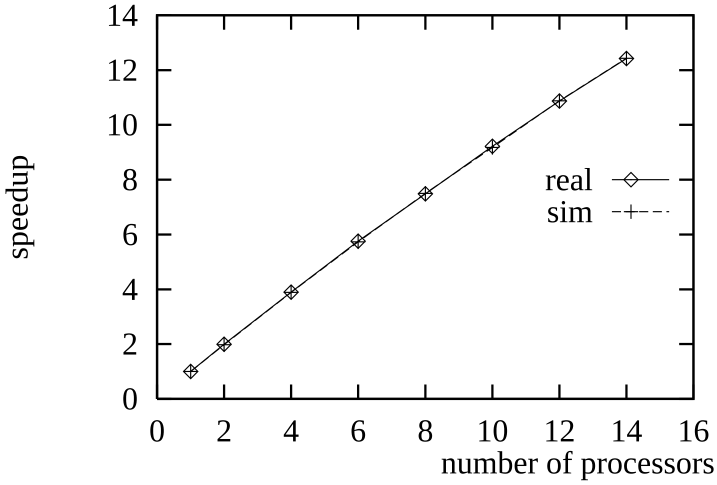

Figure 8 Actual and Simulated Speedups on the Sequent Symmetry - GAUSS

Figures 10 and 11 compare simulated and actual speedups on the Sequent for the Fast Fourier Transform program (FFT) using 8192 points. While the predicted speedups follow the actual speedups fairly closely, the errors can be seen to be more erratic. This may be attributed to the structure of the FFT program. The program creates a tree of processes with each parent waiting for its children to complete before doing its processing. Since process creation under the Sequent Operating System could not be simulated in detail, the associated overhead was approximated with an estimate, contributing to some of the inaccuracies. FFT also does a fair amount of floating point computation. The amount of time taken by each floating point instruction is highly data dependent, and since instruction execution time is predicted at compile time, this also results in some prediction errors. The percentage error in speedup prediction is once again much lower than that for execution time prediction.

Figures 12 and 13 compare predicted (sim) and real speedups for the Mergesort program (MERGE) sorting 16384 integers. The structure of this program is similar to the FFT,

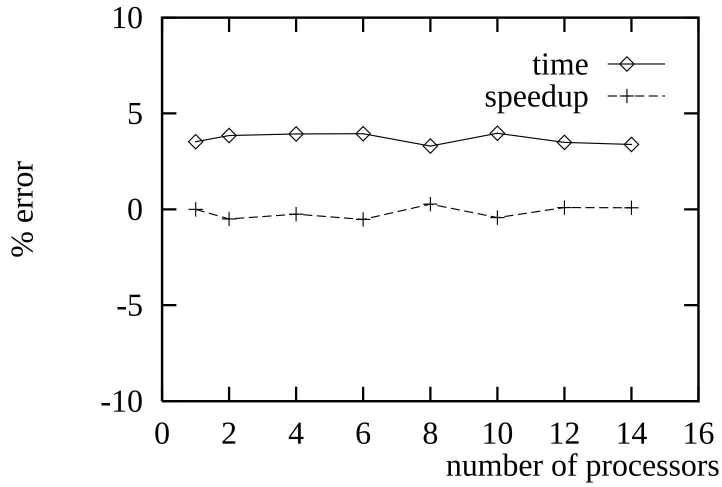

Figure 9 Errors in Prediction on the Sequent - GAUSS

resulting in poor speedups. The inaccuracies may be attributed to the estimation of process creation overhead.

Figures 14 and 15 make a similar comparison for the Subgraph Isomorphism program (ISO). Subgraph Isomorphism takes two graphs and finds all instances of one in the other.

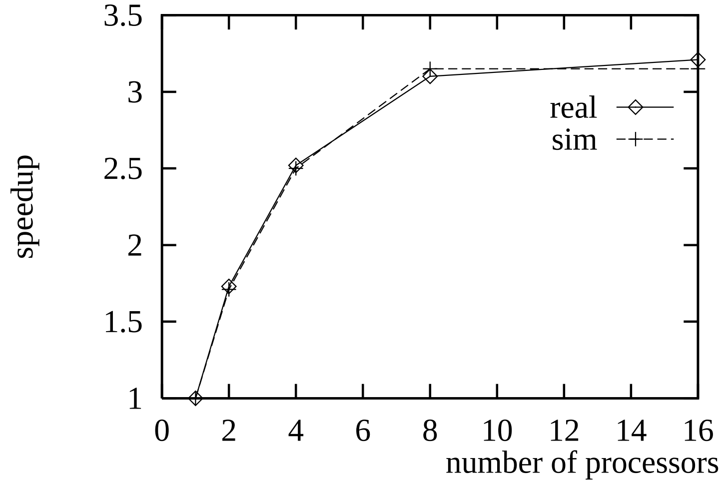

Figure 10 Actual and Simulated Speedups on the Sequent Symmetry - FFT

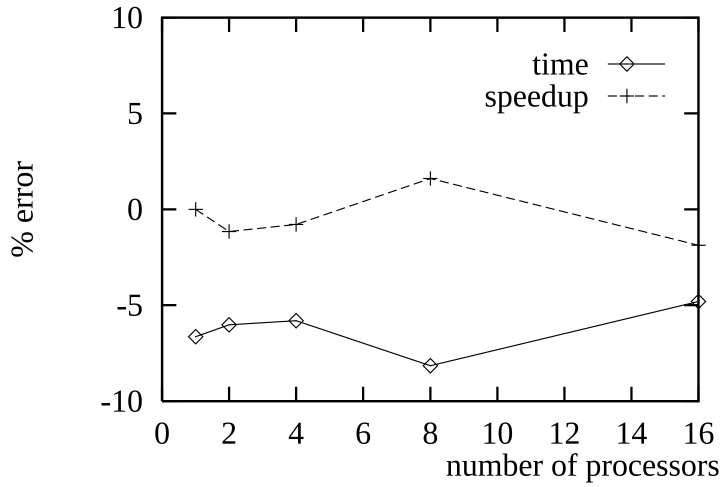

Figure 11 Errors in Prediction on the Sequent - FFT

The number of nodes in the graphs used were 3 and 20 respectively. The solution resulted in 6840 isomorphisms. The errors in timing estimation are within 5% for a simulation of up to 4 processors and the errors in speedup estimation are within 1%. The errors increase slightly as the number of processors is increased. This is because the program uses blocking

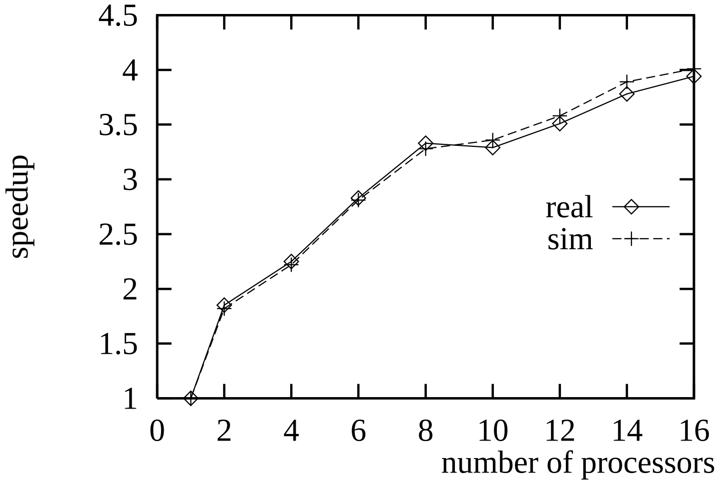

Figure 12 Actual and Simulated Speedups on the Sequent Symmetry - MERGE

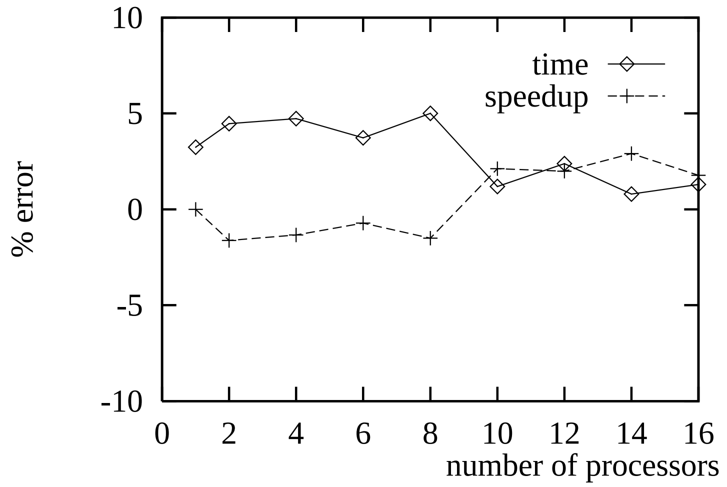

Figure 13 Errors in Prediction on the Sequent - MERGE

locks. The overhead of the resulting context switches could only be approximated and not accurately simulated.

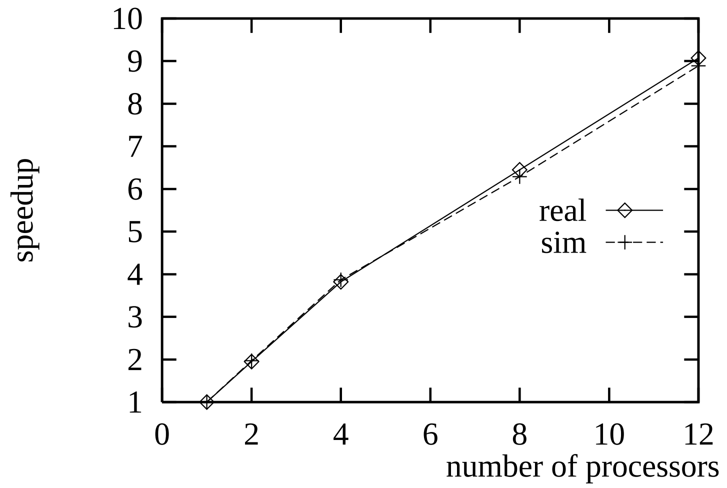

Figure 14 Actual and Simulated Speedups on the Sequent Symmetry - ISO

The percentage errors in estimation are fairly constant for all the programs as the number of processors is increased, validating the model of the Sequent architecture, and indicating that contention is modeled accurately. The percentage errors in timing estimation are within

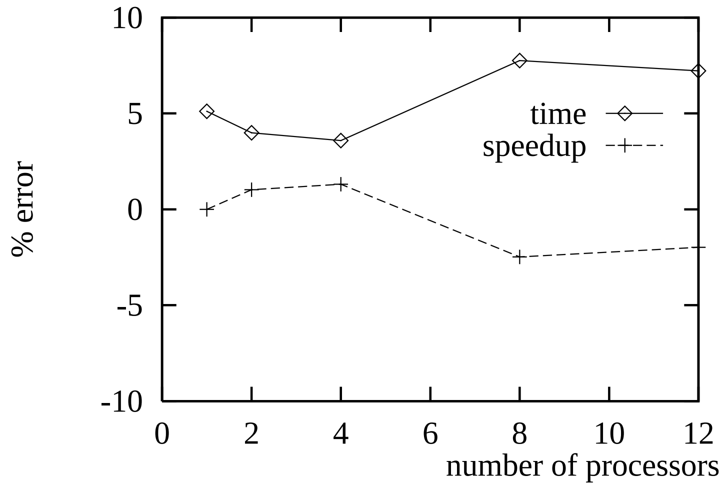

Figure 15 Errors in Prediction on the Sequent - ISO

$\pm 5\%$ in most cases. Inaccuracies may be attributed to errors in real time measurement, and the fact that the time taken by certain instructions (especially floating point instructions) is highly data dependent. In the interests of efficiency, the time taken by these data value dependent instructions is averaged, and hence results in prediction errors. The percentage errors in speedup estimation are even lower, and are within $\pm 2\%$ in most cases.

Table 1 shows the overheads incurred in simulating the different programs. Instruction-driven simulation incurs overhead in this range merely for address trace generation. The overhead incurred by the execution-driven simulation is, however, heavily dependent on the degree of detail of the architecture being simulated, as well as the number of interaction points (the degree of sharing, the number of shared variables, and the number of cache misses). The use of a split transaction bus also increases simulation overhead, since every read request on the bus results in two bus transactions. Mergesort incurs the largest overheads since it does not contain floating point code, and therefore performs less computation between interaction points (since floating point instructions take longer to execute on average). Mergesort, however, has less coherence traffic than the other programs simulated, and hence does not incur some of the overhead they do in terms of bus traffic simulation. Subgraph Isomorphism, which is another program with no floating point code, has much smaller shared data structures resulting in a smaller coherence overhead. It therefore incurs less overhead than the Mergesort program.

Program	Average Overhead
Subgraph Isomorphism (ISO)	639x
Fast Fourier Transform (FFT)	505x
Mergesort (MERGE)	861x
Gaussian Elimination (GAUSS)	598x

Table 1 Simulation Overheads - Sequent Symmetry

3.2 Single Bus SPARC Architecture

We also compared the simulator to a cycle-level simulator by using both to simulate a single-bus-based architecture of SPARC [26] processors. The cycle-level simulator used was a modified version of MPSAS [16, 17], which is a single-bus multiprocessor simulator developed by Sun Microsystems. A cycle-level simulator provides a detailed analysis of program/architecture behavior, and hence is fairly accurate. Our goal was to evaluate any loss in accuracy as well as the gain in simulation speed when using execution-driven simulation instead of cycle-level or instruction-driven simulation.

We assumed 20 MHz processors with a 10 MHz nonsplit-transaction bus. We associated a 64 Kbyte direct-mapped cache with a 32-byte line size with each processor. The protocol used was write through with no write allocate. Shared copies were invalidated on a write. The buses used a round robin arbitration scheme, and were implemented as queues (as in the Sequent architecture model).

3.2.1 Validation

Figures 16 and 17 compare the speedup predictions of the execution-driven and instruction-driven or cycle-level simulators and plot the percentage difference between the timing and speedup predictions for the Successive Over-Relaxation program (SOR). The program data consisted of a 128x128 matrix. The percentage difference increases slightly as the number of processors is increased. This may be attributed to the different data placements in the cache, which causes significant variations in performance since we used a direct-mapped cache. As the number of processors is increased, the average miss ratio and execution times decrease, making any variations more apparent. Since data placement has a smaller effect on the performance of set-associative caches, the difference in performance caused by this effect when simulating set-associative caches would be negligible. From Figure 17 we can see that the percentage difference in speedup estimation is negligible. The overhead of the cycle-level simulator is ~6 times that of the execution-driven simulator for this program (Table 2).

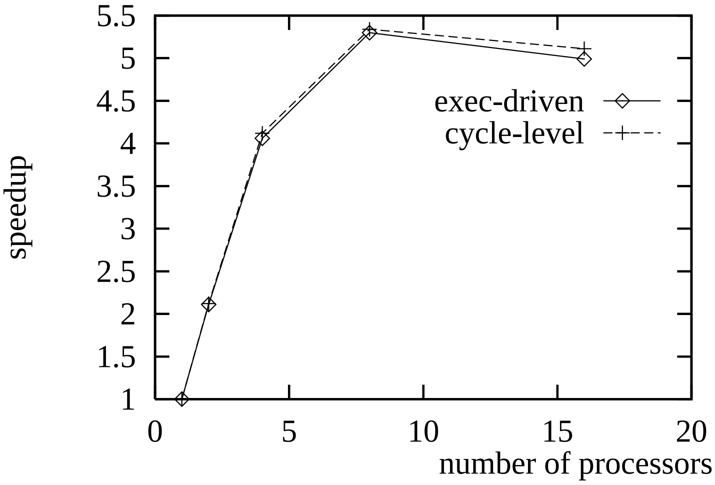

Figure 16 Execution-Driven and Instruction-Driven Speedup Predictions (SOR)

Figures 18 and 19 compare the two simulators using the Matrix Multiply (MM) program with a 128x128 matrix. The times and speedups predicted by the two simulators are almost identical, indicating that the execution-driven simulator does not lose much in accuracy. The overhead incurred by the cycle-level simulator is, however, ~30 times that of the execution-driven simulator. The reason for the difference in overheads between Matrix Multiply and SOR (Table 2) is that the execution-driven simulation incurs detailed simulation overhead only when there is activity in the system. The cycle-level simulator incurs detailed simulation overhead regardless of whether the program generates any external system activity such as bus accesses. Since the Matrix Multiply program has a very low miss rate and very little coherence traffic, its execution-driven simulation overhead is low.

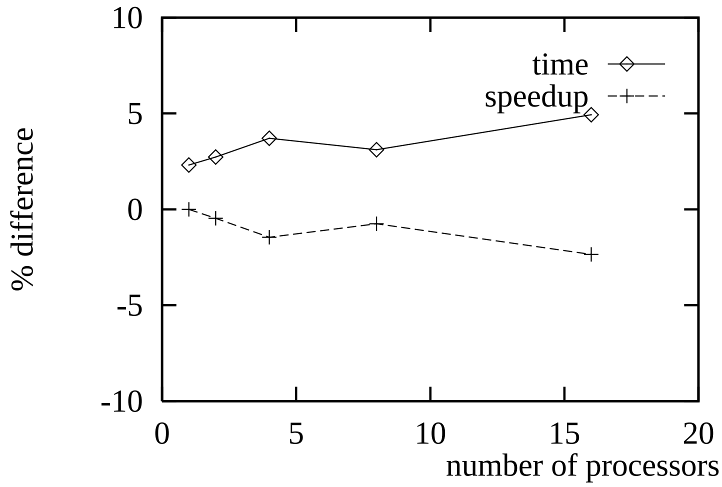

Figure 17 Percentage Differences for SOR

4 Related Work

Since the inception of execution-driven simulation ([8]), several simulation tools that use a similar concept have been developed. A similar approach to cache simulation as that used

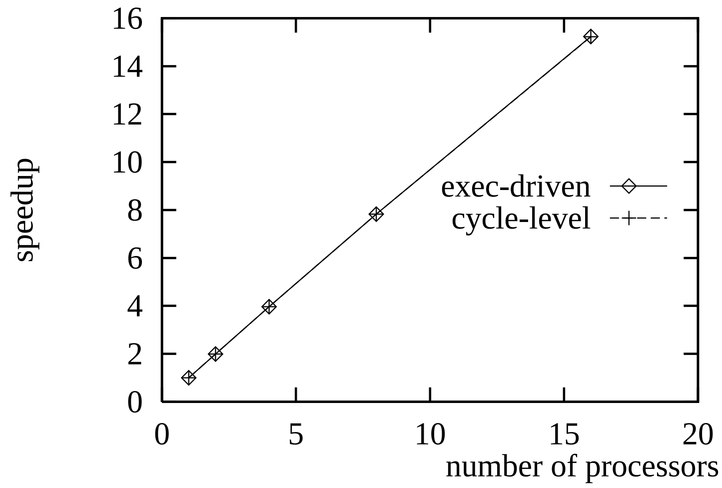

Figure 18 Execution-Driven and Instruction-Driven Speedup Predictions (MM)

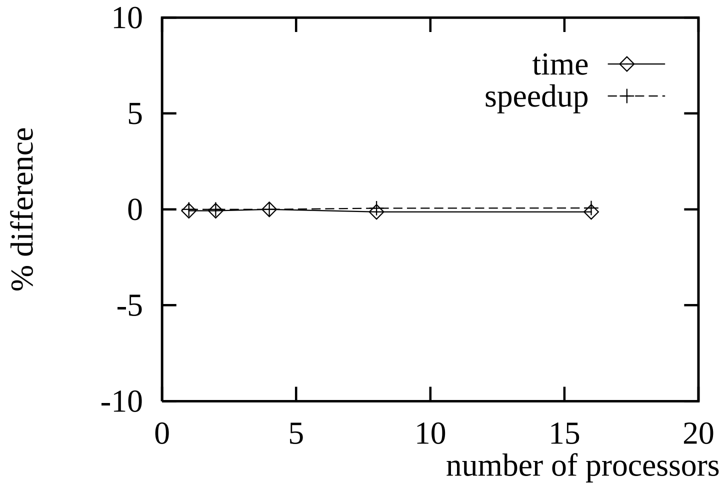

Figure 19 Percentage Differences for Matrix Multiply

Program	**Execution-Driven Overhead**	**Cycle-Level Overhead**
Successive Over-Relaxation (SOR)	**1259x**	**7682x**
Matrix Multiply (MM)	**390x**	**12740x**

Table 2 Simulation Overheads - Instruction-Driven Vs. Execution-Driven

in RPPT is used in [23], but only instruction addresses are traced. The TRAPEDS [28] approach is limited to non-multiprogrammed multicomputer address trace analysis, and does not deal with shared memory systems. TRAPEDS also does not model communication accurately, since it does not evaluate the effects of contention. Hence, it is difficult to use TRAPEDS to study trade-offs in computation and communication. We have developed and validated accurate network and communication models for both distributed and shared memory systems. The use of these models in conjunction with the execution-driven technique enables accurate performance prediction.

MPtrace [15] uses the execution-driven approach to generate traces of information that allow an address trace to be reconstructed. It achieves an overhead of 2 to 3 times normal execution time (excluding trace storage) by transferring the work of trace reconstruction to the post-processing phase. Our overhead figures include trace reconstruction, since we avoid disk access by dynamically processing the trace. The post-processing phase in MPtrace is extremely slow, generating about 3000 addresses per second [15]. Overhead reduction in the trace generation phase is also important for MPtrace since it directly affects the accuracy of the trace.

Pixie [22] is a profiling tool developed to analyze basic block usage and execution time on the MIPS R2000, and can be used to generate traces for later processing by a simulation tool. Borg et al. [3] use link time code modification to generate long traces on sequential RISC machines, once again generating traces efficiently for later analysis. ATUM [1] captures address traces using microcode. While this technique is efficient, long traces must be stored on disk, and a processor with user-programmable microcode is necessary to produce the traces. Systems of this nature do not allow for dynamic changes in the address trace due to timing variations in the architecture and timing dependent behavior of the program.

Tango [10] is a multiprocessor simulation and tracing package based on the execution-driven approach, which concentrates on program data accesses. Tango originally used UNIX processes to simulate parallelism, resulting in high context switching overhead. A recent version of Tango uses light weight processes as in our simulator. PROTEUS [5], another execution-driven simulation system, also concentrates only on shared data addresses and instruments the high-level language to generate the shared data addresses, while profiling at assembly level to extract timing information. Neither the Tango nor the Proteus projects have provided extensive validation results for shared-memory machines.

The Wisconsin Wind Tunnel [25] uses the execution-driven approach to evaluate cache-coherent shared-memory computers on a CM-5, but requires the use of the CM-5 memory's error-correcting code (ECC) bits. Mint [29] is a MIPS code interpreter for parallel programs that generates memory reference traces used to drive simulations of multiprocessor systems. Mint does not require recompiling the simulated program but suffers from the same limitations as RPPT in terms of sharing a single address space among multiple processes with possibly disjoint address spaces. Mint is also targeted only at shared memory system simulations.

The RPPT is an efficient and comprehensive execution-driven simulation tool that has been extensively validated. It allows the simulation of both message passing and shared memory multiprocessors at any desired level of detail. RPPT's modular structure allows the evaluation of a single program on several architectures or several programs on a single architecture simply by compiling in the appropriate modules.

5 Conclusions

Execution-driven simulation can be an effective performance evaluation tool for parallel architecture simulation that includes the simulation of system memory. We present results to validate the efficiency and accuracy of the technique when applied to sequential, distributed, and shared memory systems, with an emphasis on shared memory. Although the technique is not as efficient for shared-memory as it is for message-based architectures without cache/memory simulation, it is 6 to 30 times faster than cycle-level simulation. No hardware support is required and an accurately interleaved address trace is generated at run time, eliminating the necessity for large amounts of storage for traces. The simulation mimics the execution of the parallel program and simulates the architecture (including shared memory and cache organization) during execution, thereby accurately modeling the performance of an application on the architecture, including any nondeterministic behavior.

The overhead involved in execution-driven simulation of the shared-memory machines examined is low (roughly 600 times real program execution time for the entire simulation, while techniques such as instruction-driven simulation incur an overhead of the same order merely for address trace generation). The overhead can be as low as 150 in some cases. Most of the overhead is due to the detailed simulation of the architecture (network/memory/cache organization), rather than to address trace generation or code profiling. The validation results show that the timing accuracy is high (within 4% of a cycle-level simulation and within 5% for the Symmetry architecture). Predictions of relative performance metrics such as speedup tend to be even more accurate, making this technique especially attractive for comparative investigations of parallel system designs.

References

[1] A. Agarwal, R. L. Sites, and M. Horowitz. ATUM: A new technique for capturing address traces using microcode. In *Proceedings of The 13th Annual International Symposium on Computer Architecture*, pages 119–127, June 1986. vol. 14, no. 3.

[2] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. Technical Report TR 87-09-01, Department of Computer Science, University of Washington, Seattle, WA, September 1987.

[3] A. Borg, R. E. Kessler, and D. W. Wall. Generation and analysis of very long address traces. In *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990.

[4] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lus k, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, Inc., 1987.

[5] 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, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 1991.

[6] D. W. Clark. Cache performance in the VAX-11/780. *ACM Transactions on Computer Systems*, 1(1):24–37, February 1983.

[7] R. G. Covington, S. Dwarkadas, J. R. Jump, S. Madala, and J. B. Sinclair. The Efficient Simulation of Parallel Computer Systems. *International Journal in Computer Simulation*, 1:31–58, January 1991.

[8] R. G. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair. The Rice Parallel Processing Testbed. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 4–11, Santa Fe, NM, May 1988.

[9] R.G. Covington and J.R. Jump. CSIM 2.0 User's Manual. Technical Report TR 8712, Department of Electrical and Computer Engineering, Rice University, Houston, TX, October 1987.

[10] H. Davis, S. R. Goldschmidt, and J. Hennessy. Tango: A Multiprocessor Simulation and Tracing System. Technical report, Computer Systems Laboratory, Stanford University, 1990.

[11] M. Dubois, F. Briggs, I. Patil, and M. Balakrishnan. Trace-driven simulations of parallel and distributed algorithms in multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 909–915, 1986.

[12] S. Dwarkadas. Efficient methods for cache performance prediction. Master's thesis, Department of Electrical and Computer Engineering, Rice University, Houston TX, May 1989.

[13] S. Dwarkadas, J. R. Jump, R. Mukherjee, and J. B. Sinclair. Execution-Driven Simulation of Shared-Memory Multiprocessors. In *Proceedings of the 1993 SCS Western Multiconference*, January 1993.

[14] S. Dwarkadas, J. R. Jump, and J. B. Sinclair. Efficient Simulation of Cache Memories. In *Proceedings of the Winter Simulation Conference*, December 1989.

[15] S. J. Eggers, D. R. Keppel, E. J. Koldinger, and H. M. Levy. Techniques for efficient inline tracing on a shared memory multiprocessor. In *SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1990.

[16] M. Federwisch and L. Ball. *Mpsas: a Programmer and User Manual.* Sun Microsystems, 1990.

[17] J. Greenwood. The Design of a Scalable Hierarchical-Bus, Shared-Memory Multiprocessor. Master's thesis, Rice University, Houston TX, May 1992.

[18] J. R. Jump. *YACSIM Reference Manual.* Rice University, 1992.

[19] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a Cache Consistency Protocol. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 276–283. IEEE, June 1985.

[20] T. Lovett and S. Thakkar. The Symmetry Multiprocessor System. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.

[21] S. Madala. Concurrent C User's Manual. Technical Report TR 8701, Dept. of Electrical and Computer Engineering, Rice University, Houston, TX, January 1987.

[22] MIPS Computer Systems, Inc. *MIPS Languages and Programmer's Manual*, 1986.

[23] C. L. Mitchell and M. J. Flynn. A workbench for computer architects. *IEEE Design and Test of Computers*, pages 19–29, February 1988.

[24] R. Mukherjee. Efficient Simulation of Shared Memory Parallel Systems. Master's thesis, Rice University, Houston TX, May 1990.

[25] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1993.

[26] ROSS Technology, Inc. *SPARC RISC User's Guide*, second edition, Feburary 1990.

[27] Sequent Computer Systems, Inc. *Sequent Symmetry Technical Summary*, August 1987.

[28] C. B. Stunkel and W. K. Fuchs. TRAPEDS: Producing traces for multicomputers via execution driven simulation. In *Proceedings of the 1989 ACM International Conference on Measurement and Modeling of Computer Systems*, pages 70–78, May 1989.

[29] J. E. Veenstra. Mint Tutorial and User Manual. Technical Report 452, Department of Computer Science, University of Rochester, Rochester, NY, May 1993.

[30] J. P. Weinberger. Cheap dynamic instruction counting. *AT&T Bell Laboratories Technical Journal*, 63(8), October 1984.