An Address Trace Generator for Trace-Driven Simulation of Shared Memory Multiprocessors

Frank Lacy

Master's Report
Computer Science Division
University of California, Berkeley

ABSTRACT

This paper presents an extension to the standard trace-driven simulation procedure that allows for the examination of parallel programs on parallel architectures. To demonstrate the procedure, an example simulation is performed to investigate the changes resulting from modifying the architecture of the Sequent multiprocessor. The trace-driven simulation process is shown to be a very lengthy task, and other methods of predicting performance are explored.

March 7, 1988

An Address Trace Generator for Trace-Driven Simulation of Shared Memory Multiprocessors

Frank Lacy

Master's Report

Computer Science Division

University of California, Berkeley

1. Introduction

Trace-driven simulation is an important technique in the evaluation of many aspects of computer systems, especially in the analysis of memory systems. Trace-driven simulation bases its results on actual workloads, and allows for easy design prototyping [Smit85]. In the design of shared memory multiprocessors, with their multiple levels of storage and dependencies on program behavior, trace-driven simulation is very attractive. However, previous address tracers suffered from the inability to trace multiple tasks, so parallel programs could not be traced. This paper describes a multiprocess address tracer which forms the basis for extending the trace-driven simulation method to multiprocessors.

The Sequent Balance 12000 is a shared memory multiprocessor with a global 14-Mbyte memory and 12 NS32032 processors, each with a dedicated 8-Kbyte cache plus floating-point and memory management units [Mayb84]. For maximum processing power, programs on the Sequent may consist of multiple processes, with each process executing on a separate processor. Using this technique, up to a twelve-fold increase in execution speed over comparable uniprocessor performance can be realized. Of the several parallel programs developed for the Sequent from research into parallel CAD tools, three were selected as subjects for tracing.

Simulations of the Sequent architecture are run to examine the changes in performance that result from varying the ratio of processing speed to memory speed, and to compare the simulation results to performance estimates obtained from multiprocessor models. These models require orders of magnitude less time and effort than trace-driven simulation, but have not been tested against real parallel programs, only against uniprocessor programs replicated multiple times to approximate true parallel programs.

2. Software

The trace-driven simulation process occurs in two phases, generation of the address trace and then simulation of the proposed architecture(s). Four programs are involved, starting with the trace generator and ending with the multiprocessor simulator. In between are the postprocessor and cache filter. A diagram of the multiprocess trace-driven simulation procedure for three processes is shown in Figure 1. This entire process must be conducted for each target program to be traced.

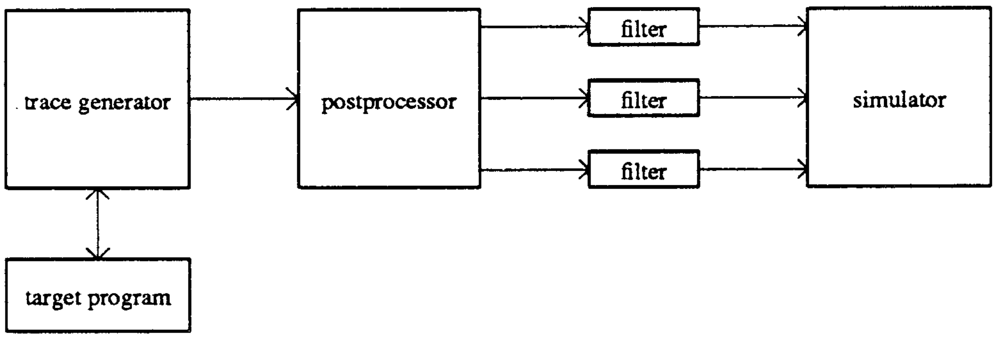


Figure 1 - Flow of data in the Multiprocess Trace-Driven Simulation procedure.

The first step, trace generation, is performed on the Sequent multiprocessor. An instruction tracer employing the Trace-bit technique of address trace generation [Agar86] is executed once for each target program. The target program to be traced (*subject*) is controlled and monitored by the tracing program (*tracer*) through the *Ptrace* system call. *Ptrace* provides the capability to start, stop, and single-step process execution and examine and modify process state. In order to support multiprogramming, the *Ptrace* facility has been extended to allow control over a complete multiple-process program, including the initial process and all of its descendants.

Traces are generated by *tracer* single-stepping the execution of *subject*. The instruction stream of *subject* is processed one instruction at a time. For each instruction, *subject* single-steps the execution of the instruction and then *tracer* interprets the instruction. The output of *tracer* is a stream of records, one record for each instruction traced. An example output record is shown in Figure 2. Each record consists of three parts: the process id, a copy of the instruction that was executed, and a series of memory accesses. Each memory access consists of an access type (data read, data write, or instruction read), a length (1, 2, or 4 bytes), and an address.

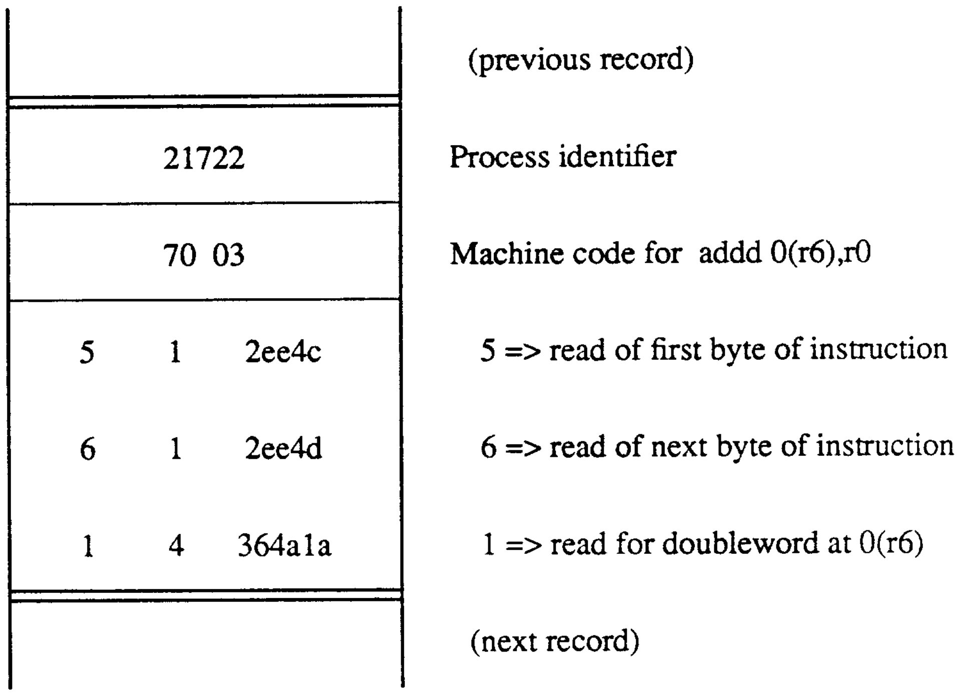

(previous record)
21722
Process identifier
70 03
Machine code for addd 0(r6),r0
5 1 2ee4c
5 => read of first byte of instruction
6 1 2ee4d
6 => read of next byte of instruction
1 4 364a1a
1 => read for doubleword at 0(r6)
(next record)

Figure 2 - Sample output record from trace generator.

Tracer monitors fork and exit calls in *subject*, so that *subject* may dynamically vary its number of processes. When *subject* consists of more than one process, *tracer* cycles through the processes of *subject*, each process executing one instruction per cycle. By setting breakpoints in *subject*, *tracer* can skip over sections of the application code, thereby limiting the trace to only those portions of *subject* deemed interesting.

Address tracing using the Trace-bit technique allows trace generation without the need for hardware or microcode modification. There is no limit to the number of instructions that can be traced, and, as mentioned before, the target program may consist of multiple, independent tasks. However, this method suffers from several drawbacks. It does not trace operating system kernel code or I/O activity, causing omission distortion, and it does not take interrupts or context switching into consideration. The tracing program also causes a large time distortion. Target programs experience a slowdown of over 100,000x compared to normal operation.

The second step of the trace procedure is trace postprocessing, where a number of small tasks take place. The binary instruction trace is converted into text, formatted with one memory reference per line, using an extended Dinero format [Hill83]. An example section of postprocessed output is shown in Figure 3. Each line includes a code specifying instruction or data reference, address of the reference, and a flag indicating shared or private reference. Additionally, for instruction references, a copy of the basic instruction (opcode and addressing modes) and the number of cycles needed to execute the instruction are included. The postprocessor also expands the one trace with interleaved references for N processes into N separate traces, one for each process. The postprocessor keeps statistics for several activities, including counts of instructions, counts of instruction references, and counts of data references, noting read or write, shared or private, and user or system.

```
2 2ee4c 4 0 8 7003      !instruction fetch at address 2ee4c;
                              shared, user; 8 cycles to execute;
                              instruction is  addd 0(r6),r0.
0 364a1a 4 0            !operand read at address 364a1a;
                              shared, user.
1 8d1ce 0 0             !operand write at address 8d1ce;
                              private, user.
1 100c44 4 1            !operand write at address 100c44;
                              shared, system.
```

Figure 3 - Sample output taken from a postprocessed file (comments added).

The postprocessor performs several functions with regard to locks, the fundamental unit of synchronization in the Sequent. All locking and unlocking activity called for by the target program is monitored. Every lock and unlock request is recorded in a special file. Counts are kept for the number of locks, number of unlocks, number of instructions executed while busywaiting for a lock, and the number of instructions executed and memory locations referenced between locks and unlocks (see Table A.5 in Appendix A). The postprocessor watches for certain illegal locking conditions, such as attempting to lock a lock that is already locked, unlocking a lock that is already unlocked, or unlocking a lock that was not previously locked. Also, the postprocessor identifies the beginning and ending points of lock and unlock sequences in the output traces.

The third step of the trace-driven simulation process is cache filtering. Cache filtering reduces the amount of trace data involved in the simulation [Puza85]. The cache filter program performs a cache simulation, separating the input records into cache hits and cache misses. The filter eliminates records from the input that could not cause bus operations, passing the other records unchanged to the output. For the write-through with invalidation coherency protocol of the Sequent, the filter passes all write references, all shared references, and all read miss references; read accesses that are cache hits are encoded in a more compact notation, where one runlength is output for each series of consecutive hits. Recorded in each runlength are the number of references compacted into the runlength, the number of cycles associated with any instructions references in the runlength, and the number of consecutive reads in the runlength, broken down into instruction references and data references. The cache filter is run once for each process recorded in the trace. The filter parameters, cache size and line size, are set according to the architecture under consideration. For a system using a cache with line size L and at least N sets, the reduced trace contains only those references that produce misses on a direct-mapped cache with N sets and line size L [Ston87].

Extending the trace-driven simulation procedure to parallel programming requires both a multiprocess address tracer and a multiprocessor simulator. Fortunately, the simulator used in this project had been previously developed by Susan Eggers for a study of data sharing in parallel programs [Egge88]. It is a complete multiprocessor simulator, modeling the processing elements, the caches and cache controllers, and the system bus. It is based on a deterministic event-driven simulator. The input to the simulator is the complete set of postprocessed/filtered traces for a target program, and the output is a file of statistics for the simulation. A file of runtime parameters determines the configuration of the architecture under consideration, setting the cache configuration, coherency protocol, memory speeds, and simulation length.

3. Theoretical Background

As an alternative to the lengthy trace-driven simulation process, Gibson has established two models for predicting shared memory multiprocessor performance, a 4-point bound (4PB) and a simple queueing model (SQM) [Gibs87]. Both models characterize a N processor machine with two parameters, $t_{compute}$ and $t_{transfer}$, the average computation time between successive main memory requests and the average memory service time, respectively. They are determined by cache reference rate, cache request ratio, and memory transfer times. The 4PB and SQM estimate three metrics of multiprocessor operation, effective uniprocessors: bus utilization, and average wait on memory. Effective uniprocessors (EU), the potential speedup of the system, indicates the maximum possible improvement in execution speed of the N processor machine over that of a uniprocessor. EU ranges from 1, no speedup, to N, linear speedup. Bus utilization (BU) is the fraction of the time that the system bus is active, with 1 representing a saturated bus. Average wait on memory (AW) is the average time between a request for memory service and the corresponding response from the memory, including the memory access time and the bus contention time.

The 4-point bound gives a region of operation bounded by an optimistic estimate and a pessimistic estimate. In the optimistic estimate there is no contention for the bus, as processors schedule their memory requests to minimize bus waiting. In the pessimistic estimate all processors issue memory requests in groups, resulting in memory contention. On average, a processor waits for half of the other processors before receiving memory service in the pessimistic estimate. For simplicity, the 4PB uses a two section piecewise-linear approximation of the pessimistic estimate. Additionally, a paranoid bound was established as a true lower bound on multiprocessor performance. In the paranoid bound, for each reference a processor must wait for all of the other $N-1$ processors to access memory before receiving memory service. The general shape of the 4PB for the EU metric is shown in Figure 4.

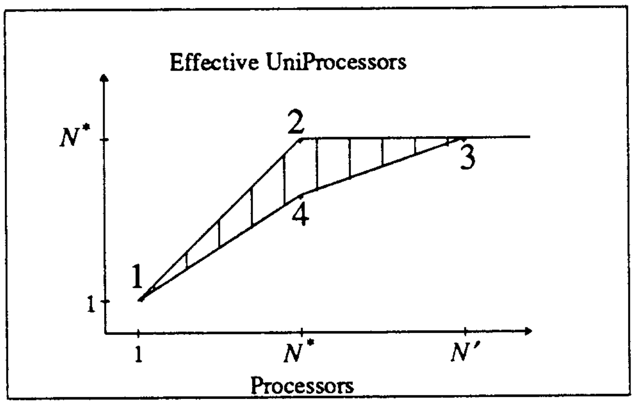


Figure 4.
The region of operation formed by the 4-point bound.
Taken directly from [Gibs87].

Computation time and memory service time determine two important points of operation of the 4PB. The system saturation point, N^*, is the saturation point of the optimistic model, and is also the maximum number of effective uniprocessors. N', the saturation point of the pessimistic model, is the number of processors necessary to insure system saturation. They are given by:

$$N^* = \frac{t_{compute} + t_{transfer}}{t_{transfer}}$$

$$N' = \frac{2t_{compute} + t_{transfer}}{t_{transfer}}$$

Also, $N' = 2N^* - 1$. These points divide multiprocessor performance into three regions of operation, $1 \le N \le N^*$, $N^* \le N \le N'$, and $N \ge N'$. The equations for the optimistic and pessimistic estimates of the 4PB for EU, BU, and AW are shown in Table 1. One additional parameter is introduced, ρ, the ratio of $t_{transfer}$ to $t_{compute}$. The 4PB predicts that when $N < N^*$, the bus is not saturated, and linear speedup is theoretically possible. The bus reaches 100% utilization using between N^* and N' processors. When $N > N'$, the 4PB predicts that the bus is fully saturated, and no speedup is possible. The equations for the paranoid bound and the nonlinear (unsimplified) pessimistic estimate are given in Table 2.

Table 1 - 4PB estimates

	optimistic		pessimistic	
	$1 \le N \le N^*$	$N \ge N^*$	$1 \le N \le N^*$	$N \ge N^*$
EU	N	N^*	$\frac{1}{3}[(2-\rho)N+(1+\rho)]$	$\frac{1}{3}[(1+\rho)N+(\frac{1}{\rho}-\rho)]$
BU	N/N^*	1	$\frac{1}{3}\rho[(2-3/N^*)N+1]$	$\frac{1}{3}[\rho N+(1-\rho)]$
AW	t_{trans}	$Nt_{trans}-t_{comp}$	$\frac{N+1}{2}t_{trans}$	$\frac{N+1}{2}t_{trans}$

Table 2 - Additional 4PB estimates

	nonlinear pessimistic	paranoid bound
EU	$\frac{N(t_{comp}+t_{trans})}{t_{comp}+\frac{N+1}{2}t_{trans}}$	$\frac{N(t_{comp}+t_{trans})}{t_{comp}+Nt_{trans}}$
BU	$\frac{Nt_{trans}}{t_{comp}+\frac{N+1}{2}t_{trans}}$	$\frac{Nt_{trans}}{t_{comp}+Nt_{trans}}$
AW	$\frac{N+1}{2}t_{trans}$	Nt_{trans}

The simple queueing model utilizes a basic queue structure with exponential distributions for memory request and service times, and provides a single estimate of performance for each of the three metrics. Two additional parameters are introduced, L_N, the average number of processors waiting on memory, and p_n, the proportion of time that there are n processors waiting on memory, with p_0 giving the bus idle time. The estimates of performance are given in Table 3.

Table 3 - Simple Queueing Model estimates

EU	$(N-L_N)\frac{t_{compute}+t_{transfer}}{t_{compute}}$
BU	$1-p_0$
AW	$\frac{L_N t_{compute}}{N-L_N}$

4. Traces

Traces were generated for several CAD programs developed for the Sequent in the EECS department at Berkeley. For this report, three traces are examined. The first is PVERIFY [Ma87], a logic verification program written by Tony Ma. PVERIFY compares two logic circuits, determining if they perform equivalent functions. The second trace is CELL [Caso86], a cell placement program by Andrea Cassotto that performs simulated annealing for IC design. The final trace is PDSPLICE [Jaco87], a circuit simulator designed by George Jacobs.

All of the target programs examined used the same method of parallelism. Programs consist of a group of identical processes, with each process repeatedly removing a segment of work from a global job queue and then processing on the segment. The traces were limited to only those sections of code with parallel processing, and for each trace the trace generator was run until six million references per process were collected. The number of child processes used by the target is fixed at time of tracing. The traces used in this paper all have the full twelve processes.

One change to all of the target programs was necessary. In order to track sharing, the postprocessor must know the addresses of all shared variables. The addresses of static and global variables can be determined from symbol tables taken from the object files, but this does not cover dynamically allocated shared variables. For these, each program to be traced was modified to record at runtime the address and size of each shared variable at time of allocation. This information was dumped into an allocation history file.

The trace generator is a very slow program. The average tracing rate is about four million instructions per day, amounting to only 300,000 instructions per process per day for a full twelve process trace. The number of instructions traced is determined by the number of processes in the trace and the number of memory references needed per process. For six million references per process, a total trace of 42 million instructions is needed, and requires from ten days to two weeks to collect. The tracer's slowness stems from its reliance on *Ptrace*. Each instruction that is interpreted requires multiple calls to *Ptrace*, and each *Ptrace* system call generates substantial overhead from the necessary context switching between the tracer and the operating system and between the operating system and the subject.

5. Simulations and Results

As an example of the complete trace-driven simulation process, simulations were run to determine the performance improvement that could be obtained from the substitution of faster processing elements into the Sequent. Simulations were performed over a range of computation speeds, with the cycles needed to execute the complete trace giving the performance indication. For each target program, six simulations were run, starting with a simulation of the Sequent architecture as it presently exists. Further simulations were conducted for faster processing speeds, with compute speed increases ranging from x2 to x32, doubling at each step.

The multiprocess trace-driven simulation process requires a large amount of both CPU time and disk space. Because of the length of time necessary to run the programs and the large sizes of the inputs and outputs, each section of the process is run separately, with the intermediate data stored on disk and tape. The sizes of the various files for the three traces are shown in Table 4, and the times for execution of the programs are given in Table 5. Trace generation is performed on the Sequent, postprocessing and filtering on a Sun 3, and simulations are run on a VAX 8800.

Table 4 - Sizes of Trace Files (in megabytes)

	compressed trace	postprocessed	filtered	filtered/ compressed
PVERIFY	306.8	981.4	325.5	55.7
CELL	334.5	1039.3	475.5	83.6
PDSPLICE	185.4	1209.9	435.9	50.8

Table 5 - Program Execution Time (in CPU hours)

	Sequent	Sun 3		VAX 8800
	trace generator	postprocessor	12 filters	7 simulations
PVERIFY	122.9	33.3	6.1	52.5
CELL	140.8	36.0	8.8	55.1
PDSPLICE	150.2	41.7	9.4	64.6

The parameters of the simulator were set to match the Sequent, 12 processors, each cache 8-Kbyte, 8 byte block, 2-way set associative, write-through with invalidation. The Sequent bus delivers a sustained data transfer rate of 8 bytes every 300 nSec [Fiel84], so $t_{transfer}$ was set to 3 cycles (100 nSec cycles time). The simulation results are based on 100,000 references per process. To overcome cold start effects, the simulator was run until steady state cache performance was obtained, then statistics were collected over the next 100K references. Also, simulations at 300K references per process were run for the x1 configuration.

The results of the simulations are shown in Figures 5 through 10, with the raw data given in Appendix A. There are eight graphs per subject. The x axis in each graph is the increase in computation speed. For each subject, the first two graphs show $t_{compute}$ and ρ. The next two graphs show the cycles needed to execute the trace and the speedup, the ratio of the cycles for the base machine (x1) to the cycles for the faster processor. The next three graphs show the values of EU, BU, and AW, along with the various predicted values. The final graph shows the value of N^* and N', the saturation points.

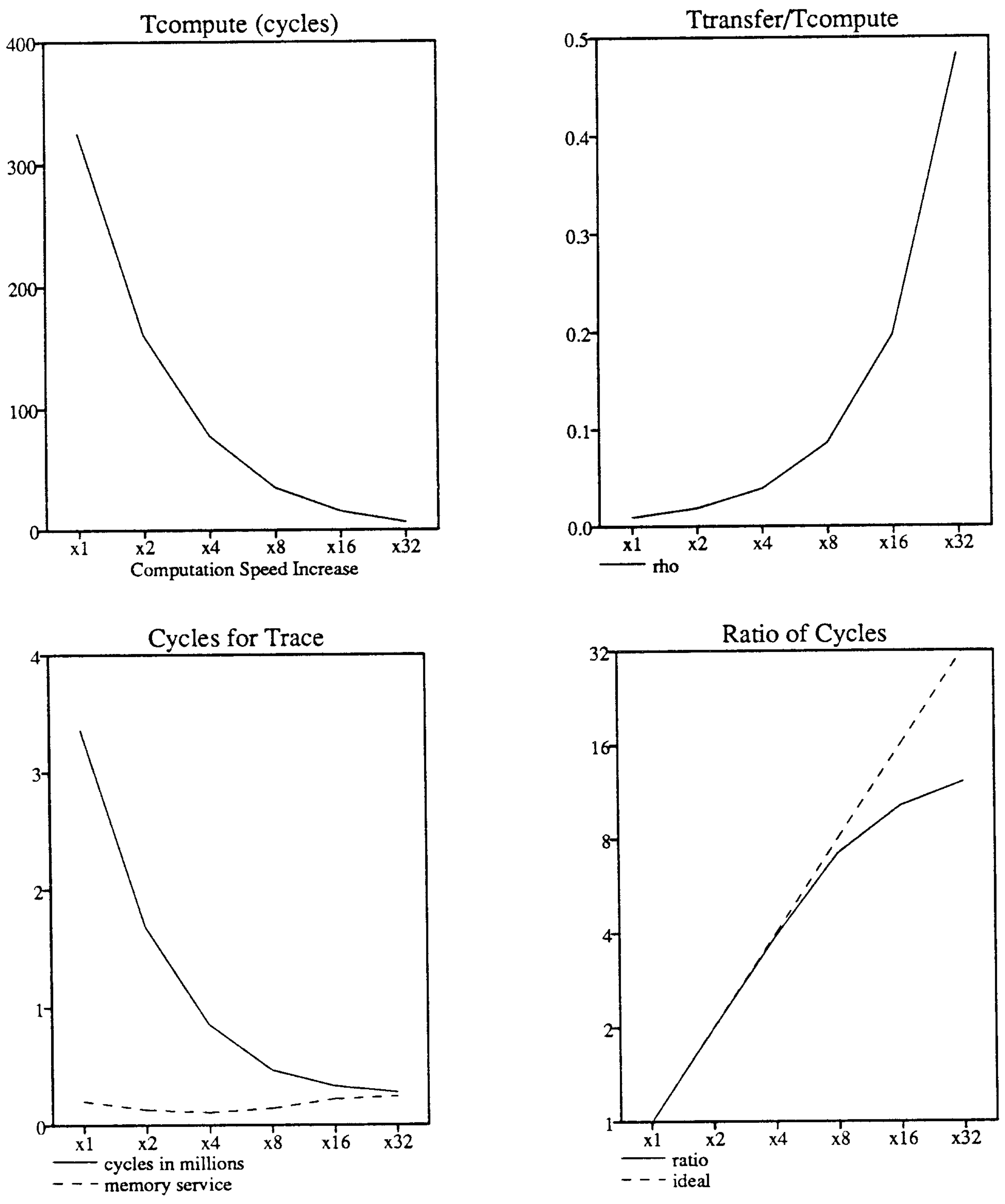


Figure 5: PVERIFY Simulation Results

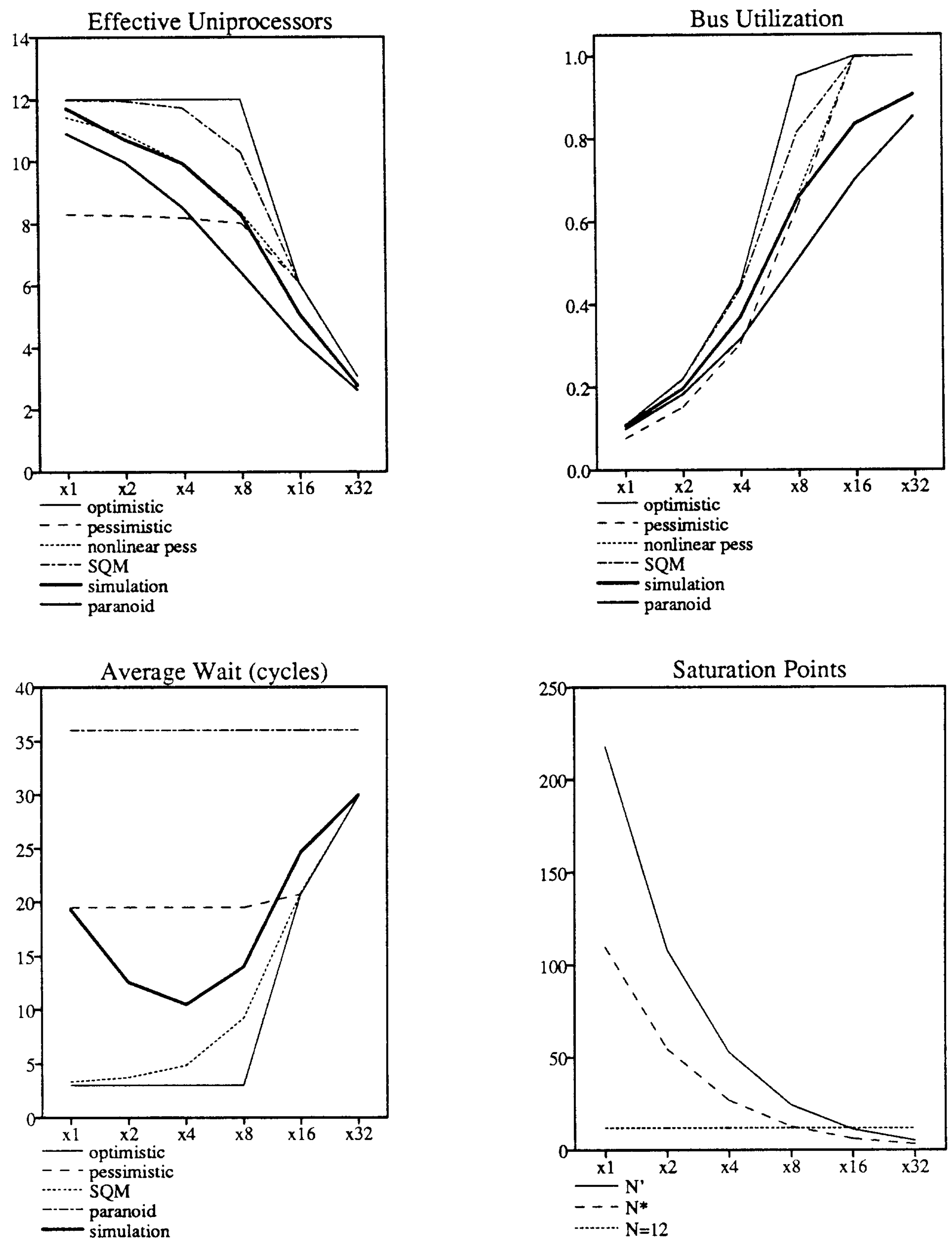


Figure 6: PVERIFY Performance Metrics

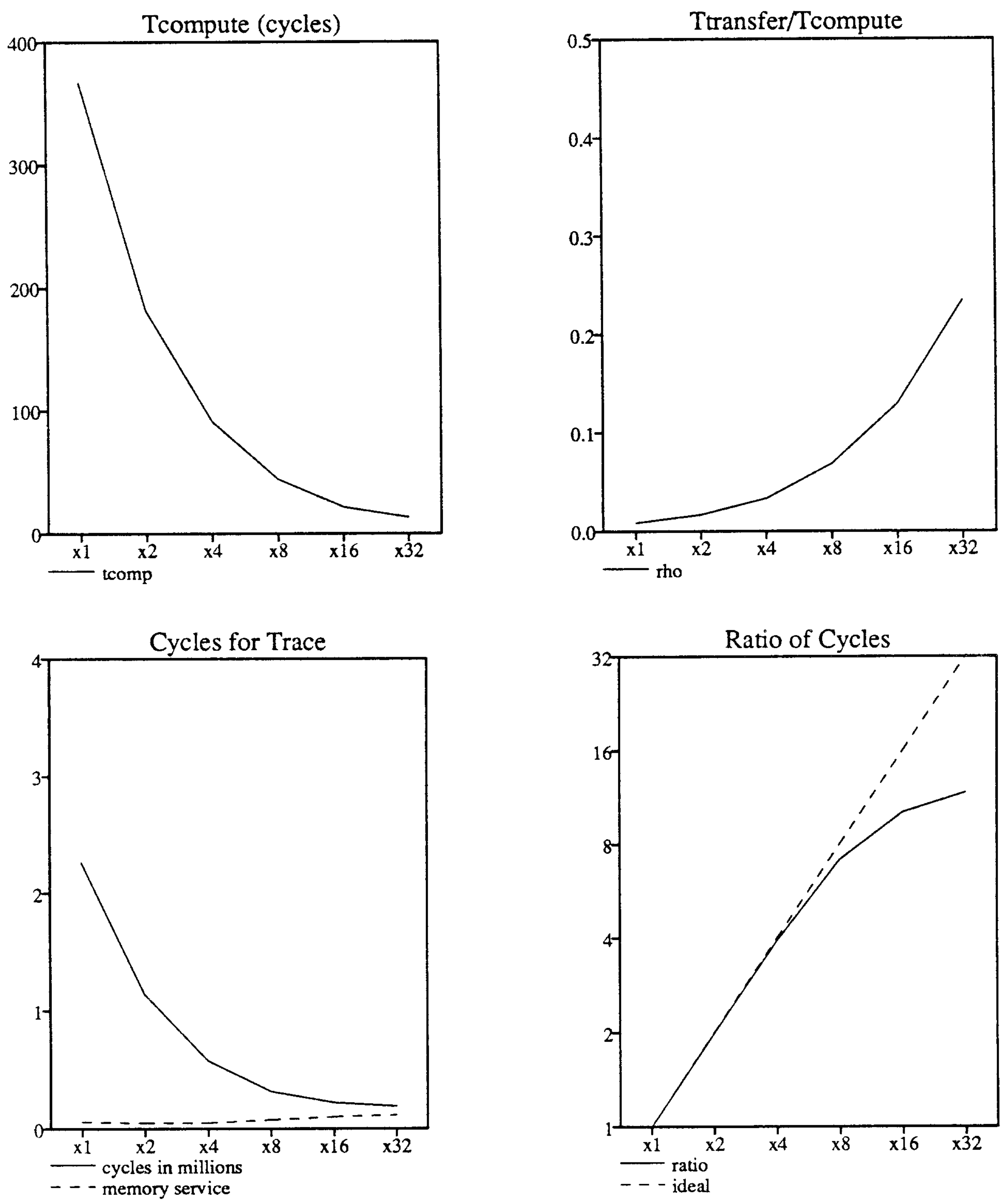


Figure 7: CELL Simulation Results

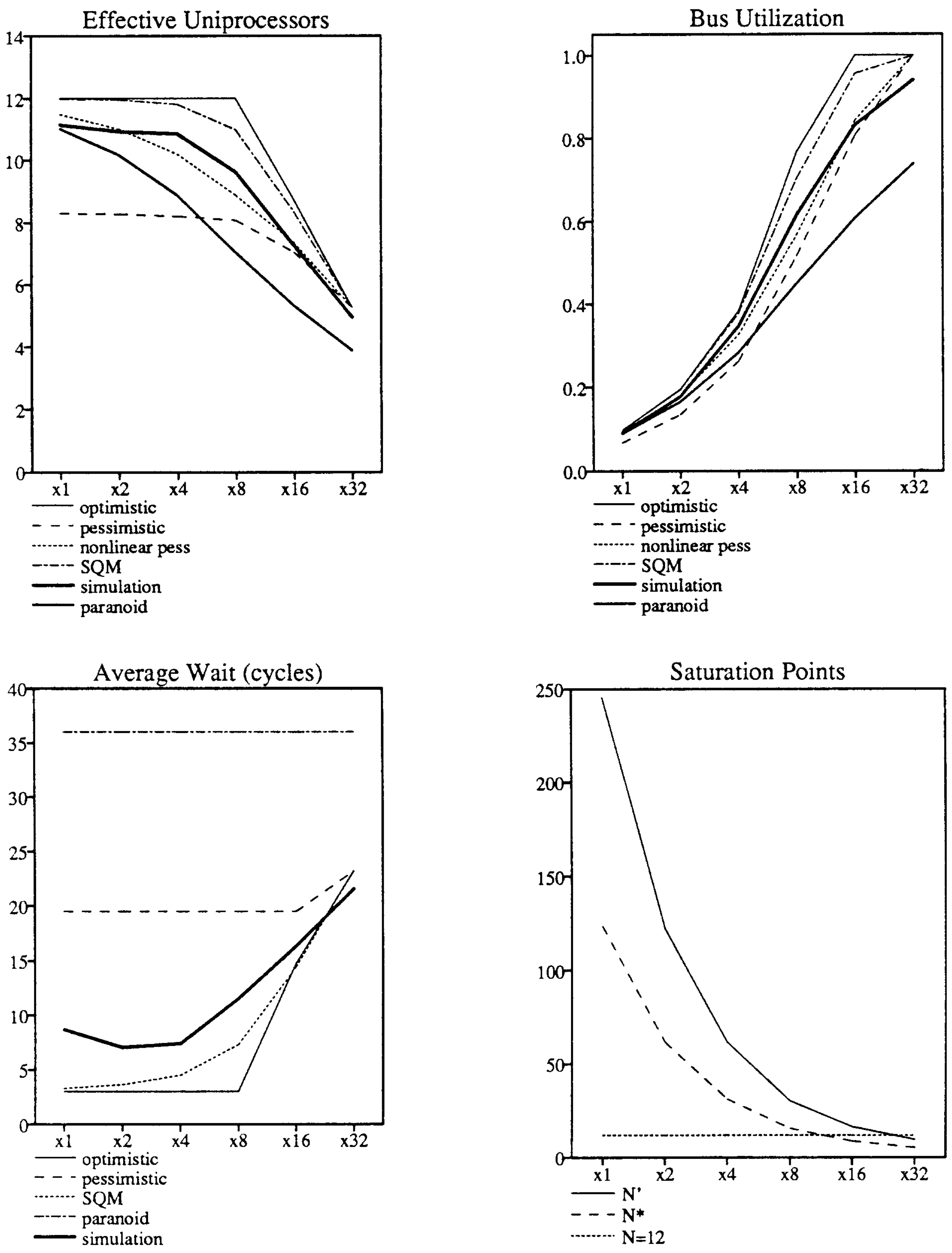


Figure 8: CELL Performance Metrics

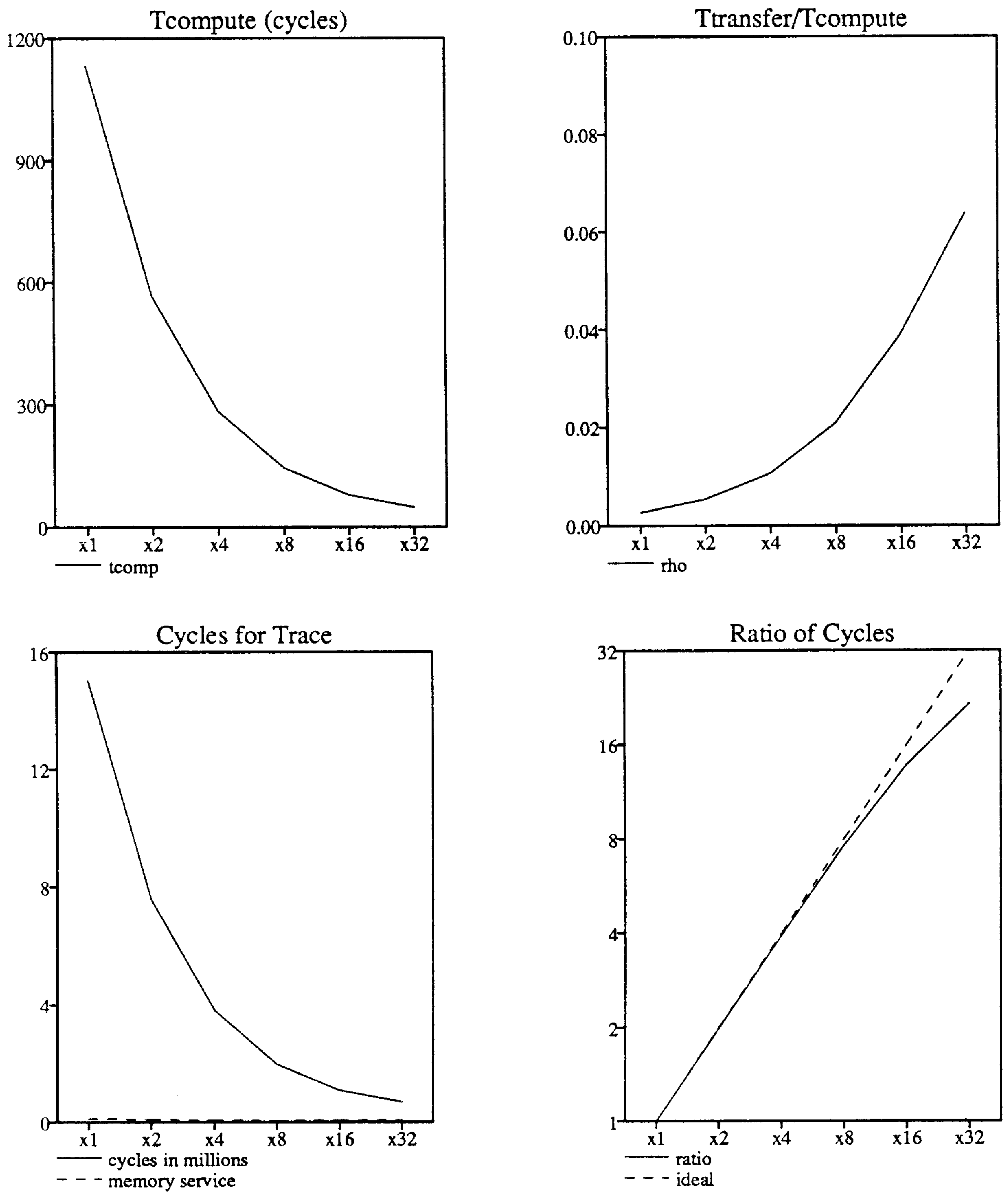
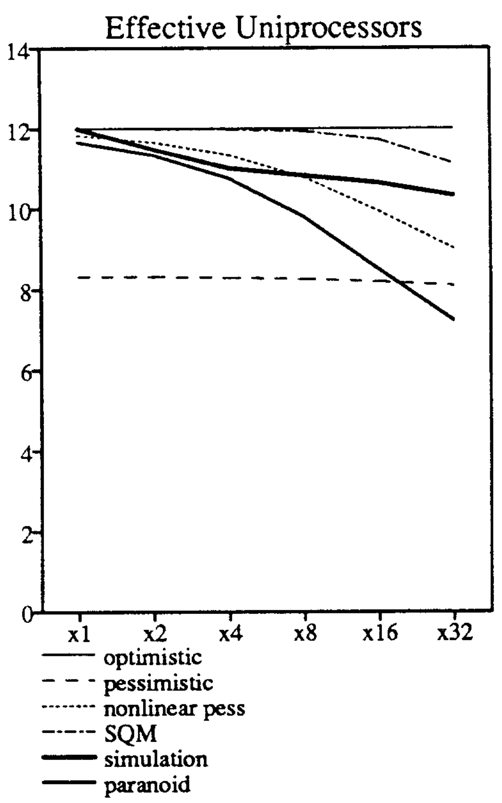
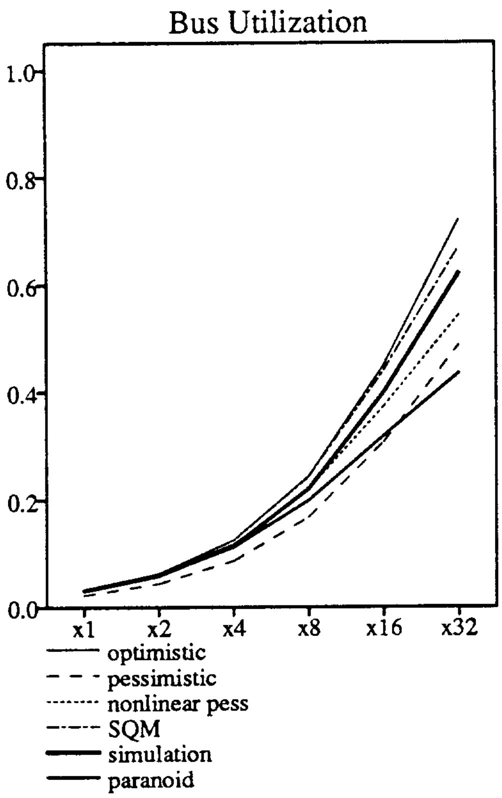
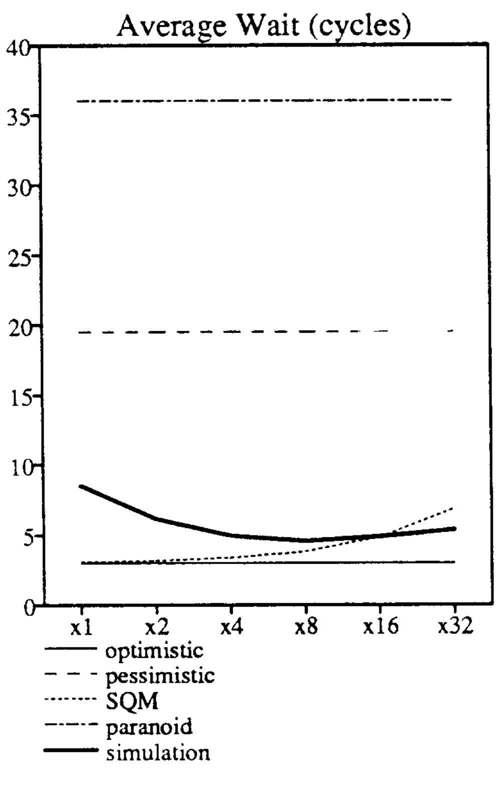


Figure 9: PDSPLICE Simulation Results







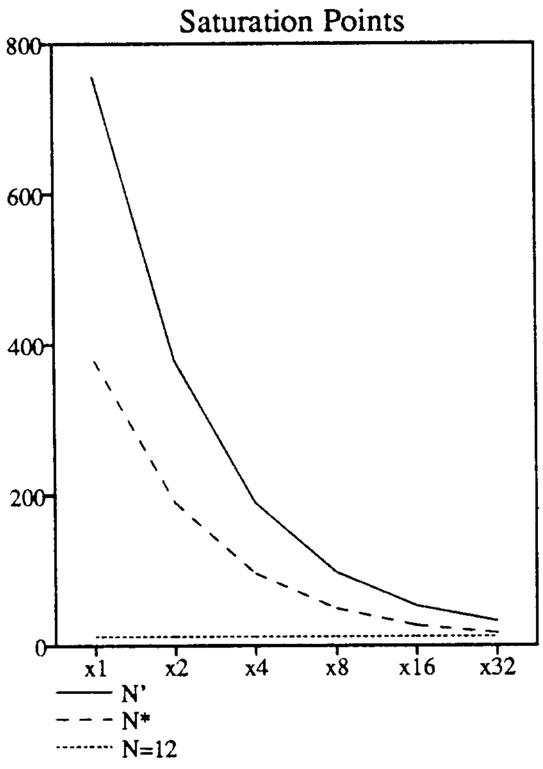


Figure 10: PDSPLICE Performance Metrics

6. Discussion

Overall, the results show that a substantial increase in performance is possible. PVERIFY and CELL show very similar results. In the x1 configuration, the values of $t_{compute}$ are large in comparison to $t_{transfer}$, and the values of ρ are a very low 0.01. The bus is underutilized - approximately 10% BU, and the EU figures are close to 12. Execution time is dominated by compute cycles, leaving much room for improvement. In both programs, execution speed nearly doubles between the x1 and x2, between the x2 and the x4, and again between the x4 and x8 simulations. The x16 simulations are the first to have $\rho > 0.1$, and the first to have $N^* < 12$. In the x16 and x32 simulations, the bus utilization approaches 1.0, and the effective uniprocessors drop off significantly. The x32 simulations are definitely in the knee of the curves showing the ratio of cycles, and little additional speedup would be available from further increases in computation speed. The PDSPLICE simulation shows an even larger value for $t_{compute}$, four times that of PVERIFY and CELL. This extremely high value allows for nearly linear speedup up to x32. Even at the fastest speed, the bus is still underutilized ($N^* > 12$ for all of the PDSPLICE simulations). PDSPLICE would require at least another doubling of processor speed before the bus would reach saturation.

Results from the longer 300K simulations agree fairly closely with the shorter runs: the values of $t_{compute}$ and N^* are within 5%, and the values of AW and EU are within 8%. The bus utilization metric shows the greatest difference, with the BU from the longer simulations being 10% lower than the short simulations; this difference occurs for all three target programs.

Comparisons with the theoretical models show that the performance of the 4PB is similar to that reported in Gibson's work, with the 4PB bounding the performance metrics, except near N', where the EU and BU curves cross under the pessimistic estimates. Performance is consistently lower than that predicted by the SQM, and the error of the SQM relative to the simulations can be as high as 25%. This is quite different from the earlier work using uniprocessor traces, in which the predictions of the SQM were within 3% of the simulation results. In general, the simulation curves lie approximately halfway between the optimistic estimates and paranoid bounds. For $N < N'$, the nonlinear pessimistic estimate is very close to the simulated results for both the EU and BU curves. In all but one case, the difference between the simulation and the nonlinear pessimistic estimate is less than 8% for both the EU and BU.

In Figure 11, the optimistic and pessimistic estimates of the 4PB form the more traditional graphs of the metrics versus the number of processors. Two simulations are shown, PVERIFY x1 and PVERIFY x8. The x1 graphs indicate that the base architecture could theoretically be expanded to include up to 100 processors, with speedup ranging form 75 to 100. The x8 graphs show more typical behavior. Bus saturation occurs with ten to twenty processors.

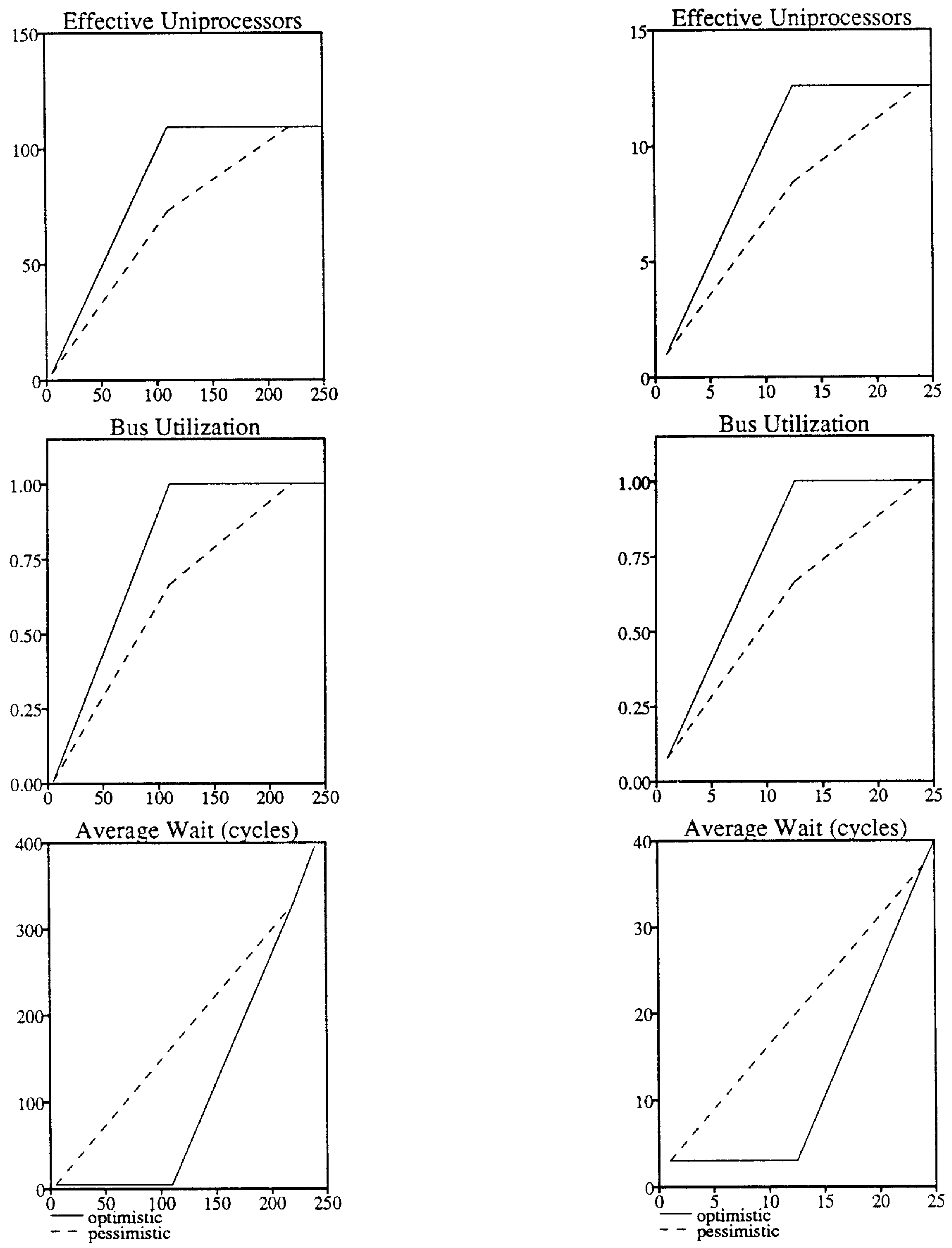


Figure 11: 4PB Estimates for PVERIFY, x1 on left and x8 on right.

One surprising result is the tendency for the AW to decrease as the processing speed is increased. This is quite visible in the AW curves, especially the AW curve for PVERIFY. Intuitively, AW should increase with decreasing $t_{compute}$. AW is determined by two quantities, L_N, from the SQM, and λ, the average rate of arrival of bus requests; the relationship is given by $L_N = \lambda AW$. To investigate this AW behavior, the simulator was modified to provide times for the start and completion of each bus request. From the start/completion times, p_n (from the SQM) can be determined. L_N is related to p_n by:

$$L_N = \sum_{n=1}^{N} n p_n$$

The quantity λ is simply the total number of bus operations divided by the number of cycles for the trace.

This modified simulator was used to rerun the PVERIFY x1 and the PVERIFY x4 simulations. Although the computation speed increases by a factor of four between these two simulations, the AW of the faster processor is approximately one-half that of the slower processor. The graphs of p_n are shown in Figure 12, and the values of L_N, λ, and AW are given in Table 6. This method of calculating AW agrees very closely with the AW taken from the simulator output. The value of L_N for PVERIFY x4 is about twice that of PVERIFY x1, while the value of λ for PVERIFY x4 is about four times that of PVERIFY x1, resulting in a decrease in AW from PVERIFY x1 to PVERIFY x4.

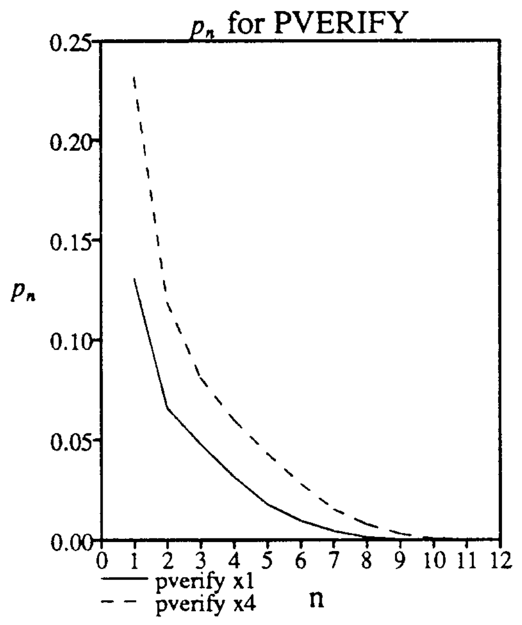


Figure 12 - Graph of p_n - the fraction of time that there are n processors waiting on memory. The horizontal axis shows n, the number of processors.

Table 6 - Calculation of AW

trace	L_N	λ	L_N/λ	AW simulator
PVERIFY x1	0.7231	0.0375	19.29	19.30
PVERIFY x4	1.5391	0.1466	10.50	10.49

The results of PVERIFY x1, very high $t_{compute}$ and very low **BU, yet high** AW, lead to the supposition that bus requests from the processors occur at roughly the same time, intermixed with long periods of idle bus. In each of the target programs examined, all of the processes making up the program execute the same code, starting at the same place in the program at the same time. The paths of the various processes can diverge due to branches and locks, but the processes will reconverge at any synchronization barriers in the code. The contention for the bus during the periods of references drives the AW up, while the long idle periods keep the BU low.

This behavior is shown in Figure 13. The first example (a) shows perfectly matched $t_{compute}$ values, resulting in no memory contention. Any variation of $t_{compute}$ between processors disturbs the staggered lock-step pattern. In (b), one value for $t_{compute}$ does not match the others. The third processor issues its second memory request two cycles early, resulting in two cycles of waiting, thereby increasing the average wait. In (c), the $t_{compute}$ value is divided by two (x2 faster processor). The third processor's second reference still causes memory contention, but now the wait has been increased by only one cycle, instead of two. Thus, the AW metric is lower for the faster processor, since the AW depends only on the number of cycles spent accessing memory, not on the percentage of time spent accessing memory. In both (b) and (c), after the second group of requests, the processors are lined up again in perfect staggered order for the next group of requests. An actual example taken from the PVERIFY x1 simulation is shown in (d). In (e), this example from PVERIFY has been modified with a x4 speedup, and the result is fewer cycles for waiting, decreasing from 20 to 8, while the number of cycles on the bus has remained constant at 30. This decrease in AW can continue until the start of the next series of requests occurs before the end of the present series, in which case the overlap will cause a sharp increase in AW. Each series of requests can last up to $Nt_{transfer}$, so the AW can decrease only while $t_{compute} > Nt_{transfer}$. In the PVERIFY simulations, $t_{transfer}$ is less than $Nt_{compute}$ starting with the x8 simulation. The x8 simulation is also the first to have an increasing

AW. Thus, in this model, the decrease in AW is a result of the decrease in the inter-process variance of $t_{compute}$ that comes from the scaling of $t_{compute}$.

```
     oBooooooooooooooooBooooooooooooooooBooooo
     ooBooooooooooooooooBooooooooooooooooBoooo
(a)  oooBooooooooooooooooBooooooooooooooooBooo
     ooooBooooooooooooooooBooooooooooooooooBoo
     oooooBooooooooooooooooBooooooooooooooooBo

     oBooooooooooooooooBooooooooooooooooBooooo
     ooBooooooooooooooooBooooooooooooooooBoooo
(b)  oooBooooooooooooooxxBooooooooooooooooBooo
     ooooBooooooooooooooooBooooooooooooooooBoo
     oooooBooooooooooooooooBooooooooooooooooBo

          oBooooooooBooooooooBoooo
          ooBooooooooBooooooooBooo
     (c)  oooBoooooooxBooooooooBoo
          ooooBooooooooBooooooooBo
          oooooBooooooooBooooooooB

     oBBBoooooooooooooo  ...241...  ooBBBooooooooooooo
     oxxxBBBooooooooooo  ...241...  ooooooooBBBooooooo
(d)  oooxxxxBBBoooooooo  ...241...  ooooxBBBoooooooooo
     oooooooooxBBBooooo  ...241...  oooooooooxxBBBoooo
     oooooooooxxxxBBBoo  ...241...  oooooooooxxxxxBBBo

     oBBBoooooooooooooo  ...48...  ooBBBoooooooooooooo
     oooxBBBooooooooooo  ...48...  ooooooBBBoooooooooo
(e)  ooooooxBBBoooooooo  ...48...  oooooooxxBBBooooooo
     ooooooooooBBBooooo  ...48...  oooooooooooxBBBoooo
     ooooooooooooxBBBoo  ...48...  oooooooooooooxxBBBo
```

--- LEGEND ---

o - compute cycle ...n... - series of n compute cycles

x - waiting on memory B - accessing memory

Figure 13 - Examples of AW:
(a)-no contention at x1 (b)-contention at x1, 2 wait cycles
(c)-contention at x2, 1 wait cycle (d) example from PVERIFY x1, with 20 wait cycles
(e)-PVERIFY example with x4 speedup, showing 8 wait cycles

7. Summary

Multiprocessor trace-driven simulation is seen to be a very useful but very arduous procedure. The simulation results verify the accuracy of the 4-point bound, but the predictions of the simple queueing model are always too optimistic. The nonlinear pessimistic estimate forms the best single estimate of performance, predicting within 10% of the simulations. As in the previous work, the quantity ρ, the ratio of $t_{transfer}$ to $t_{compute}$, is found to be the most important factor in predicting performance. For the three CAD programs examined, the base Sequent architecture shows a very low value for ρ, allowing for a significant increase in performance by exploiting faster processors. For future studies, more work needs to be done in examining the average wait metric versus processor speed. The pattern of references made by the processors should be studied to measure the correlation between the processors and to measure the distribution of the $t_{compute}$ values. Also, it would be interesting to perform the simulations with copy-back caches, rather than write-through.

8. Acknowledgments

Thanks to Susan Eggers, who directed my research from start to finish and also wrote the simulator, and Garth Gibson, who provided help and suggestions for this paper. Special thanks to Dave Patterson, who served as both my research advisor and major field advisor. Thanks to all of the people of the SPUR project for all of the resources, especially the tremendous amount of disk space, that I was able to use. Also, thanks to Alberto Sangiovanni-Vincentelli and Richard Newton for use of the Sequent. This material is based upon work supported under a National Science Foundation Graduate Fellowship.

9. Bibliography

[Agar86] A. Agarwal, R. L. Sites, and M. Horowitz, "ATUM: A New Technique for Capturing Address Traces Using Microcode", *Proceedings of the 13th International Symposium on Computer Architecture*, June, 1986, pp 119-127.

[Caso86] A. Casotto, F. Romeo, and A. Sangiovanni-Vincentelli, "A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells", *IEEE International Conference on Computer-Aided Design*, November, 1986, pp 30-33.

[Egge88] S. J. Eggers, "Simulation Analysis of Data Sharing Support in Shared Memory Multiprocessors", Ph.D. dissertation, University of California, Berkeley, 1988.

[Fiel84] G. Fielland and D. Rogers, "32-bit computer system shares load equally among up to 12 processors", *Electronic Design*, September, 1984, pp 153-168.

[Gibs87] G. Gibson, "Estimating Performance of Single Bus, Shared Memory Multiprocesors", Master's Report, University of California, Berkeley, 1987.

[Hill83] M. D. Hill, "Evaluation of On-Chip Cache Memories", Master's Report, University of California, Berkeley, 1983.

[Jaco87] G. K. Jacobs, A. R. Newton, and D. O. Pederson, "Parallel Linear Equation Solution in Direct-Method Circuit Simuators", *IEEE International Symposium on Circuits and Systems*, May, 1987, pp 1056-1059.

[Ma87] H. T. Ma, S. Devadas, R. Wei, and A. Sangiovanni-Vincentelli, "Logic Verification Algorithms and their Parallel Implementation", *Proceedings of the 24th Design Automation Conference*, July, 1987, pp 283-290.

[Mayb84] W. Mayberry and G. Efland, "Cache Boosts Multiprocessor Performance", *Computer Design*, November, 1984, pp 133-138.

[Puza85] T. R. Puzak, "Cache-Memory Design", Ph.D. dissertation, University of Massachusetts, 1985.

[Smit85] A. J. Smith, "Cache Evaluation and the Impact of Workload Choice", *Proceedings of the 12th International Symposium on Computer Architecture*, June, 1985, pp 64-73.

[Ston87] H. S. Stone, *High-Performance Computer Architecture*, Addison-Wesley, 1987.

10. Appendix A: Raw Data

This appendix presents the actual figures from the simulations and the various performance estimates. The first table gives the data output by the simulator, and the following three tables show the predictions for the three traces. The pessimistic estimate is defined only for $N \leq N'$. The column labeled nonlinear is the nonlinear pessimistic estimate, and only the EU and BU estimates differ from the pessimistic. The last table gives statistics taken from the postprocessor. The statistics are for the entire trace. Shown are the number of lock variables declared in the target, number of calls to the *lock* library routine, number of instructions and number of memory references in the trace, and the number of instructions executed while busywaiting for a lock.

Table A.1 - Results From Simulations

trace/speedup	t_{comp}	ρ	N^*	N'	Kcycles	EU	BU	AW
PVERIFY x1	325.36	0.0092	109.45	217.90	3361	11.71	0.107	19.30
PVERIFY x2	160.59	0.0187	54.53	108.06	1688	10.72	0.197	12.56
PVERIFY x4	77.55	0.0387	26.85	52.70	856	9.94	0.370	10.49
PVERIFY x8	34.89	0.0860	12.63	24.26	467	8.28	0.656	14.01
PVERFIY x16	15.27	0.1965	6.09	11.18	330	5.09	0.836	24.68
PVERIFY x32	6.22	0.4826	3.07	5.14	277	2.78	0.906	29.94
CELL x1	366.61	0.0082	123.20	245.41	2263	11.15	0.091	8.69
CELL x2	181.62	0.0165	61.54	122.08	1139	10.93	0.178	7.05
CELL x4	90.92	0.0330	31.31	61.61	577	10.86	0.347	7.40
CELL x8	43.91	0.0683	15.64	30.27	317	9.63	0.616	11.54
CELL x16	21.22	0.1292	8.74	16.48	223	7.27	0.832	16.35
CELL x32	12.78	0.2347	5.26	9.52	193	4.95	0.941	21.60
PDSPLICE x1	1132.37	0.0026	378.46	755.91	15046	12.00	0.032	8.52
PDSPLICE x2	566.64	0.0053	189.88	378.76	7567	11.38	0.060	6.15
PDSPLICE x4	283.96	0.0106	95.65	190.30	3828	11.02	0.115	4.96
PDSPLICE x8	144.45	0.0208	49.15	97.30	1975	10.83	0.220	4.58
PDSPLICE x16	77.18	0.0389	26.73	52.45	1090	10.65	0.398	4.91
PDSPLICE x32	46.99	0.0638	16.66	32.33	696	10.34	0.620	5.39

Table A.2 - PVERIFY Performance Estimates

trace/speedup		optimistic	pessimistic	nonlinear	paranoid	-SQM-	simulation
	EU	12.00	8.30	11.43	10.90	11.99	11.71
PVERIFY x1	BU	0.110	0.076	0.104	0.100	0.110	0.107
	AW	3.00	19.50		36.00	3.33	19.30
	EU	12.00	8.26	10.90	9.99	11.95	10.72
PVERIFY x2	BU	0.220	0.152	0.200	0.183	0.219	0.197
	AW	3.00	19.50		36.00	3.73	12.56
	EU	12.00	8.19	9.96	8.51	11.73	9.94
PVERIFY x4	BU	0.447	0.305	0.371	0.317	0.437	0.370
	AW	3.00	19.50		36.00	4.85	10.49
	EU	12.00	8.02	8.36	6.41	10.31	8.28
PVERFIY x8	BU	0.950	0.635	0.662	0.508	0.816	0.656
	AW	3.00	19.50		36.00	9.22	14.01
	EU	6.09			4.28	6.06	5.09
PVERIFY x16	BU	1.000			0.702	0.996	0.836
	AW	20.73			36.00	20.88	24.68
	EU	3.07			2.62	3.07	2.78
PVERIFY x32	BU	1.000			0.853	1.000	0.906
	AW	29.78			36.00	29.78	29.94

Table A.3 - CELL Performance Estimates

trace/speedup		optimistic	pessimistic	nonlinear	paranoid	-SQM-	simulation
	EU	12.00	8.30	11.49	11.02	11.99	11.15
CELL x1	BU	0.097	0.067	0.093	0.089	0.097	0.091
	AW	3.00	19.50		36.00	3.29	8.69
	EU	12.00	8.27	11.02	10.18	11.96	10.93
CELL x2	BU	0.195	0.134	0.179	0.165	0.194	0.178
	AW	3.00	19.50		36.00	3.64	7.05
	EU	12.00	8.21	10.21	8.88	11.81	10.86
CELL x4	BU	0.383	0.262	0.326	0.284	0.377	0.347
	AW	3.00	19.50		36.00	4.49	7.40
	EU	12.00	8.08	8.88	7.04	10.99	9.63
CELL x8	BU	0.768	0.517	0.568	0.451	0.703	0.616
	AW	3.00	19.50		36.00	7.31	11.54
	EU	8.74	7.05	7.36	5.31	8.35	7.27
CELL x16	BU	1.000	0.807	0.843	0.608	0.956	0.832
	AW	14.78	19.50		36.00	4.60	16.35
	EU	5.26			3.88	5.25	4.95
CELL x32	BU	1.000			0.738	0.999	0.941
	AW	23.22			36.00	7.74	21.60

Table A.4 - PDSPLICE Performance Estimates

trace/speedup		optimistic	pessimistic	nonlinear	paranoid	-SQM-	simulation
PDSPLICE x1	EU	12.00	8.32	11.83	11.66	12.00	12.00
	BU	0.032	0.022	0.031	0.031	0.032	0.032
	AW	3.00	19.50		36.00	3.09	8.52
PDSPLICE x2	EU	12.00	8.31	11.66	11.34	12.00	11.38
	BU	0.063	0.044	0.061	0.060	0.063	0.060
	AW	3.00	19.50		36.00	3.18	6.15
PDSPLICE x4	EU	12.00	8.29	11.35	10.76	11.98	11.02
	BU	0.126	0.087	0.119	0.113	0.125	0.115
	AW	3.00	19.50		36.00	3.38	4.96
PDSPLICE x8	EU	12.00	8.26	10.79	9.81	11.93	10.83
	BU	0.244	0.168	0.220	0.200	0.243	0.220
	AW	3.00	19.50		36.00	3.83	4.58
PDSPLICE x16	EU	12.00	8.19	9.95	8.50	11.73	10.65
	BU	0.449	0.307	0.372	0.318	0.439	0.398
	AW	3.00	19.50		36.00	4.86	4.91
PDSPLICE x32	EU	12.00	8.10	9.02	7.23	11.14	10.34
	EU	0.720	0.486	0.541	0.434	0.668	0.620
	AW	3.00	19.50		36.00	6.87	5.39

Table A.5 - Statistics from Postprocessor

trace	number of instructions	number of references	number of lock variables	number of lock calls	instructions in busywaiting
PVERIFY	37,084,169	63,147,075	16	25,572	483,232
CELL	32,996,913	71,218,150	31	23,908	1,028,616
PDSPLICE	42,120,631	79,348,924	151	114,706	1,302,756