EFFICIENT SIMULATION OF CACHE MEMORIES

S. Dwarkadas
J. R. Jump
J. B. Sinclair
Department of Electrical and Computer Engineering
Rice University
Houston, TX 77251, U.S.A.

ABSTRACT

Cache memories are used in computer systems to reduce average memory access times. Existing techniques for predicting cache performance are often unsatisfactory in terms of cost or performance. This paper presents a method for efficiently simulating the effects of a cache on the execution time of a program. We use an execution-driven simulation approach that requires no hardware support and provides a highly accurate dynamic address trace to a cache simulation model. Almost all of the overhead in this approach is in the cache simulation rather than the address trace generation. The cache simulator is used in conjunction with the Rice Parallel Processing Testbed to study the performance of concurrent programs executing on multiprocessor systems with caches. We have also developed an estimative execution-driven simulator that greatly reduces the simulation overhead by using parameters extracted from a detailed simulation of a program's execution on a processor with a cache, along with an analytical model of cache behavior. The predictions and overhead of the estimative technique are compared with those obtained from detailed cache simulations.

1. INTRODUCTION

The development of improved techniques for cache performance analysis is important because existing techniques typically suffer from drawbacks such as large time and space complexity, specificity to an existing architecture, or lack of accuracy. When examining the performance of multiprocessor designs that include multiple caches, additional problems arise due to coherency enforcement and the size of the traces required for a representative analysis (which is proportional to the number of processors).

The goal of this research is to develop a technique that accurately and efficiently simulates the behavior of computer systems with cache memories. Toward this end, we have extended the *execution-driven* (Covington, et al. 1988) paradigm to support the efficient simulation of caches in uniprocessors and multiprocessors. This approach requires no hardware support and adds relatively low time and space overhead. It is based on the principle of basic block profiling (Weinberger, 1984).

The execution-driven cache simulator is used in conjunction with the Rice Parallel Processing Testbed (RPPT) (Covington, et al. 1988) in order to efficiently simulate cache-based multiprocessor designs. This enables an accurate prediction of the time at which events occur on the target multiprocessor, and prevents any shifting of their order that unpredicted delays due to cache misses might cause.

Performing a detailed cache simulation adds a significant overhead to the actual execution of the program. A simpler simulation model, which gives an estimate of the effect of the cache on the program, has also been developed. This simplified execution-driven simulator uses parameters extracted from a detailed execution-driven cache simulation of the program under study, as well as an analytical model for the cache behavior, to give a fairly accurate estimate of the performance of the system.

Existing cache performance prediction techniques include hardware measurement (Clark, 1983), trace-driven simulation (Smith, 1982, Smith, 1987), and analytical modeling (Agarwal, et al. 1989, Smith, 1987). Hardware measurement requires specialized support equipment as well as costly implementation, and limits analysis to an existing cache. Trace-driven simulation post-processes a collected address trace in order to determine the performance of different cache organizations. It has the disadvantage of being costly both in space and time. Large caches require the use of long traces, exacerbating the problem. Analytical modeling tries to characterize a trace of execution and its address reference pattern using a few parameters, which are then used to determine the performance of various cache designs. The predictions suffer from lack of accuracy due to the inability to model program characteristics correctly.

Most software-driven tracing methods use an instruction-by-instruction simulation of the application. They require some form of single-step interrupting support or specialized microcode (ATUM (Agarwal, et al. 1986)). In the former case, for every instruction executed, the state of the processor is saved, the instruction is decoded, the effective addresses of memory references are generated and passed on to the cache simulator, and finally the original state is restored for correct execution (Thiebaut and Stone, 1987). Lack of hardware tracing support or the simulation of a different CPU implies an instruction-level simulation that emulates execution by interpreting each instruction (examples are MILS (Mips Instruction Level Simulator) and TRACER (Agarwal, et al. 1986)).

2. EXECUTION-DRIVEN CACHE SIMULATION

The main goal of execution-driven simulation is to avoid the high overhead associated with instruction-level simulation of sequential computers, while retaining most of the accuracy. The resulting system is program-driven, but the overhead is significantly reduced by parsing the program to get timing estimates and address trace information (at compile time). The information obtained is then used at runtime to generate performance predictions dynamically while avoiding a detailed emulation of each instruction's execution.

The ratio of emulation time to actual workload execution time is called the *slowdown factor* or *overhead*. The overhead associated with instruction-level simulation and address trace generation is reported to be of the order of 1000 (Agarwal et al., 1986). Using hardware tracing, the overhead is about 100, and if microcoding is used, the overhead is about 10 (this figure applies to address trace generation, and not to the complete cache simulation (Agarwal et al., 1986)) . By extending the execution-driven concept to the dynamic generation of address traces and simulation of caches, we have developed a space-efficient approach that is significantly faster than conventional instruction-level simulations.

2.1 Basic Block Profiling

The assembly language program is viewed as a collection of code segments called basic blocks. A basic block is a series of consecutive assembly language instructions that is contained within no larger basic block and has the property that once the block is entered during execution, all instructions in it will be executed exactly once. Hence, accounting for the execution time of the entire basic block can be done at the beginning of the block (Covington, 1988). This approach, called basic block profiling, keeps the overhead low without sacrificing much in terms of accuracy. A similar approach is used in (Mitchell and Flynn, 1988) for the simulation of caches. However, only instruction addresses are traced. The steps necessary in order to profile a program and simulate the cache and the design of the translator are described below. The implementation and issues described here are for the 68020 CPU and the SUN Unix C compiler. The principles can, however, be extended to design a profiler for any other processor and instruction set.

Generation of Assembly File and Executable Profiling is done at the assembly language level, since the assembly code is easily modified in order to capture the runtime information required. A complete executable that corresponds to the assembly version of the file must be generated. This is necessary in order to generate accurate values for all virtual addresses in the text space of the program, since these can be statically determined at compile time.

Assembly File Parser and Profiler We have developed a one-pass profiler, which parses the input assembly program and inserts the necessary instructions to perform a dynamic cache simulation of the program. Profiling analysis requires the following steps —

1. The symbol table from the a.out executable is read to determine the address at which each subroutine begins.
2. The program is broken down into a set of basic blocks.
3. At the start of the main procedure, code is inserted to read the per basic block addresses that will be statically determined and written to a file by the profiler, as well as to initialize the data structures for the cache simulation.
4. Instructions are inserted at the start of each basic block to perform the runtime cache simulation and determine the total execution time taken by each basic block.
5. As each instruction is parsed, its instruction space address references are calculated, and, along with the time taken by the instruction, are inserted into a file (*static address trace*). The addresses are associated with the basic block to which the instruction belongs.
6. When an access by an instruction to a data reference is determined, code is inserted to extract the data address dynamically at runtime, and a marker is inserted in the static address trace to indicate the point at which the reference is made.

As the profiler parses the start of every subroutine in the assembly language program being profiled, it sets a simulated program counter to the correct value, which is read from the symbol table at the end of the a.out file. The value of the label identifying the subroutine, along with the text address base value, determines the starting address of the routine. The symbol table is initially read from the a.out file and stored as a hashed linked list in order to make the search for the value of any label quick.

For each program file, the profiler collects the following information:

- virtual instruction addresses accessed by the program,
- the number of clock cycles required for each reference, and
- a value to indicate whether the operation is a read or a write

At runtime, the code inserted at the start of every basic block reads the statically generated addresses and dynamic address markers for that basic block and determines the effects on the cache as well as on the time taken by the basic block. Addresses are read until either the end of the trace for the basic block or a marker for an address that must be dynamically generated is encountered. The code inserted before a data reference extracts the data address at runtime, and then processes the rest of the static address trace for the basic block until either another dynamically generated address or the end of the reference stream for the block is encountered. Figure 2.1 shows a piece of 68020 assembly code with profiling instructions (indented).

```
jbsr    _atoi
      movl #2, _bblkno    /*set the basic block
                           *number to the correct
                           *value*/
      jbsr _genadr        /*simulate the cache until
                           *the next data address*/
addqw   #0x4, sp
      pea a6@(-56)        /*extract the data address*/
      movl sp@+, _addr
      movl #0, _memacctype
      movl #3, _cycles    /*indicate the time
      jbsr _insert         *taken by the access*/
movl    a6@(-0x38), d0
      pea a6@(-0x18)
      movl sp@+, _addr    /*write access*/
      movl #1, _memacctype
      movl #2, _cycles
      jbsr _insert
clrl    a6@(-0x18)
moveq   #0x1, d0
```

Figure 2.1: Profiled Assembly Code

Statically Generated Addresses These addresses are normally instruction addresses since they are deterministic and do not change after having been linked into the final executable. Introduction of the profiling and simulation code changes the values of program addresses. It is essential that the address trace and timing information generated should be that which would actually occur if the original program rather than the profiled version were run. Since all instruction addresses are generated statically during profiling, this guarantees that the code inserted into the assembly language program in order to accomplish the profiling does not affect the values of the addresses generated. Hence, all instruction addresses are generated using information from the executable's symbol table.

Dynamically Generated Addresses Data addresses that are referenced indirectly either through a register or through memory cannot be generated statically. Similarly, absolute address references cannot be generated statically, even though they can be determined at compile time, since such references would be at variance with the data addresses generated at runtime that have a constant displacement due to inserted code and data, as discussed later. Hence, all data addresses are generated dynamically.

Insertion of code between existing assembly language instructions can cause errors with instruction-space related jumps. These jumps make implicit assumptions about the number of bytes between the current program counter value and the location to be read. Figure 2.2 shows an example. Any errors can be avoided simply by not inserting code before a jump statement, which is not required in most situations.

```
     cmpl #6, d0
     jhi L141
     movw pc@(6, d0:l:2), d0/*insertion of code at
                            *this point will result
                            *in the wrong address
                            *being loaded into d0*/
     jmp pc@(2, d0:w)
L142:
     .word L125-L142
     .word L128-L142
     .word L132-L142
     .word L131-L142
     .word L135-L142
     .word L133-L142
```

Figure 2.2: Example of an instruction-space related jump.

In the specific case of the SUN Unix linker and loader, the executable format consists of three logical segments —

1. the *text segment*, which starts at the beginning of the second page in virtual memory,
2. the *data segment*, which starts at the first segment boundary immediately following the text, and
3. the *stack*, which starts at the highest possible address in virtual memory, and grows downwards.

The addresses in the text segment are guaranteed to be identical to the original program's addresses since they are generated at compile time. The stack addresses, which are generated at runtime, are also unaffected, since the stack is always in the same state as the original program while it is executing its own code. The data addresses generated at runtime, on the other hand, will have a constant offset from the original addresses because of the additional instructions and data inserted by the profiler. The effect of the additional instruction segment space is minimized since the offset to the data is in integer multiples of the segment size (128 Kbytes). Offset of the data space because of the increase in the size of the instruction segment is unlikely (it does not occur in the programs used in this research because of the small size of their instruction spaces).

If the program does not use dynamic allocation, the effect of the additional data added by the profiler can also be removed. This will be guaranteed if the profiled program is the first to be linked into the executable, causing no offset to the data addresses. (In practice, however, the SUN Unix linker tries to minimize allocated space by compacting the initialized and uninitialized data segments together. This can cause a slight displacement of the uninitialized data segment due to differences in compaction of the segments in the two executables. The displacement is not significant since it is somewhat arbitrary, and causes a constant offset in the mapping of the data to the cache. This should not affect cache performance to any significant degree.) However, a large number of programs use dynamic allocation. For these programs, in order to ensure that there is no relative offset within the program's data, the profiling code must be the first to be linked into the executable, and the ensuing constant offset subtracted from the data addresses.

Assembly The profiler produces a modified assembly language listing of the original program, which may then be assembled and linked with the subroutines that perform the collection of runtime information and cache simulation. The result is a self-profiling executable image that, in conjunction with the address trace file generated during profiling, can be used to determine the cache and timing performance of the original program.

2.2 Multiprocessor Version

The profiler can also be used in conjunction with the RPPT. Currently, this allows the simulation of shared memory multiprocessors with independent caches (no caching of shared variables) or of distributed message-based systems in which each processor has its own cache and main memory. This is useful in determining the shift of synchronization points that may occur because of larger delays in data or instruction access due to more cache misses on one processor than on another. For example, if two processes sent messages to a third at approximately the same time, the order in which they are received could be interchanged due to the presence of a cache.

Independent caches of any organization on a common bus may be simulated. When a miss is encountered in simulating the cache, global time must be updated, allowing any synchronization that may be necessary due to bus access by the cache to occur. The effects of bus contention are also simulated.

2.3 Limitations and Possible Enhancements

The timing analysis performed by the profiler is not 100% accurate. Since timing analysis must be performed at compile time, instructions such as shifts and branches may introduce inaccuracies into the analysis. This is because the time required by these instructions at runtime depends on the values of the operands, quantities that are not available

during the compile time analysis of the program. In order to overcome this problem, an increased overhead is required in determining the values of the data dynamically.

Currently, timing analysis is implemented as a table-driven lookup of timing costs for each instruction, which requires a scan and parse of instructions to break them up into opcode and operands. The 68020 is a pipelined processor with an on-chip instruction cache. For each 68020 instruction the manufacturer specifies a *best-case*, *cache-case*, and *worst-case* execution cycle count. The *best-case* cost reflects the time for instruction execution when the instruction is in the on-chip cache and benefits from maximum overlap due to other instructions. The *cache-case* reflects the time when the instruction is in the cache but has no overlap. The *worst-case* reflects the time when the instruction is not in the cache, or the cache has been disabled, and there is no instruction overlap. For a given program, the timing analysis must choose which weighted average of cases best represents the typical program execution conditions.

The parser accounts for some of the pipelining effects by assuming overlapping instruction fetches within a basic block. Additional pipelining effects can be modeled to some extent by dynamically determining successful and unsuccessful branch conditions, and assigning different timing costs to them. The cost of the added accuracy of such a scheme is the extra overhead per basic block required by the status setting and checking instructions. In the current implementation of the profiler, the start of a basic block is always assumed to flush the pipeline.

A possible source of error in generating the address trace is any dependence of the number of addresses referenced on the actual values of the data. As the assembly code is analyzed at compile time, this is not taken into consideration since extracting the necessary information would involve a detailed dynamic analysis. An example of an instruction that can potentially give rise to this problem is the *move multiple register* instruction. One of the instruction's operands is either immediate (a constant) or register direct (variable, not used by the SUN Unix C compiler), and indicates which registers need to be saved or restored.

Our analysis also does not take into account any effects of prefetching on bus and cache activity (prefetching could cause references that are not actually made by the program to occur). This prevents any *implementation distortion* (Agarwal, et al. 1986) of the trace (specificity to the implemented architecture). The architecture-independent trace generated by ignoring these effects is more suitable for examining the effects of other architectural variations.

2.4 Trace Accuracy

The traces generated by the detailed execution-driven simulation were validated by comparing them with actual traces collected from the 68020 bus on a SUN–3/60 using a logic analyzer. The on-chip instruction cache was disabled in order to obtain an address trace that was not distorted due to cache prefetches. The lengths of traces obtained from the logic analyzer were 1000 addresses long, the maximum storage available on the logic analyzer. The virtual address traces were found to match exactly for the benchmarks used, which were traces collected from runs of the FFT (Fast Fourier Transform) and mergesort algorithms. The data addresses were displaced by a constant factor, which was to be expected because of the addition of data space by the profiling code.

2.5 Timing Validation

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 timing validation was performed by comparing system generated times for the execution of the unprofiled program on the SUN–3/280 with the times generated by the program's simulation. The timing measurement for the unprofiled program was performed with the help of the hardware clock and software that ensures that only user time for the process is measured.

In determining the time taken within the 68020, a mix of *best-case*, *cache-case* and *worst-case* timings have been used, namely, $0.6*cache+0.4*worst-0.2*(cache-best)$. The value of 0.6 was chosen since this was the average hit ratio quoted using the benchmarks in (Smith, 1987) for the 64–byte on-chip instruction cache of the 68020. The first two terms contribute to instruction times assuming no overlap with other instructions. In order to compensate for the pipelining effect, the difference between the cache-case and best-case timings was subtracted, presuming that the pipeline was effective 20% of the time. In addition, the profiler compensates for some of the pipelining within a basic block by subtracting the time that would have been taken if part or all of the instruction were not already fetched by the previous 4–byte instruction fetch. The 20% pipeline utilization is an experimentally derived factor chosen in order to most accurately predict, for a number of algorithms, the execution times as measured on the hardware.

Figures 2.3–2.6 compare the time predicted by the simulator (*sim*) with the time actually taken to run the algorithm on the Sun-3/280 (*real*). The validation was performed for two algorithms — the FFT and the mergesort. The FFT is floating-point intensive, while the mergesort has no floating point computation. The error can be seen to be a maximum of -4% for the FFT and a maximum of $+4\%$ for the mergesort algorithm. Since the error is almost constant, it can be attributed to pipelining and instruction cache effects that are not being modeled. The accuracy is 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.

The error in prediction by the profiler without simulation of the cache, that is, assuming that all cache accesses are hits (*sim, h=100%*), has also been plotted on the same graphs for comparison. The error increases as the hit ratio for the algorithm decreases (caused by an increase in the data size).

3. ANALYTICAL MODEL FOR CACHE MISS RATIO PREDICTIONS

Trace-driven simulation is a simple method of evaluating cache memory systems with varying hardware parameters. However, evaluation of realistic workloads is often impractical due to the enormous time overhead involved in the simulation. A simple analytical cache model for uniprocessors that provides estimates of cache performance in terms of miss

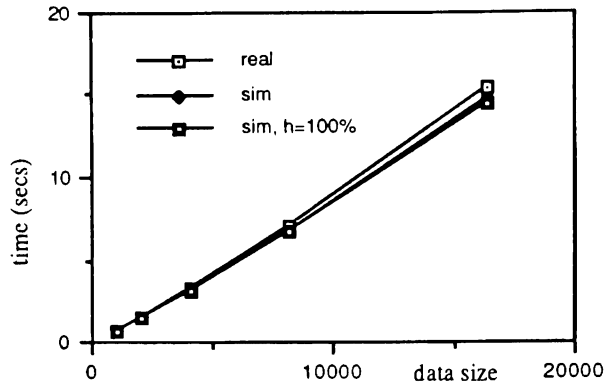

Figure 2.3: Validation of the Sun-3/280 68020 detailed cache simulator using the FFT algorithm

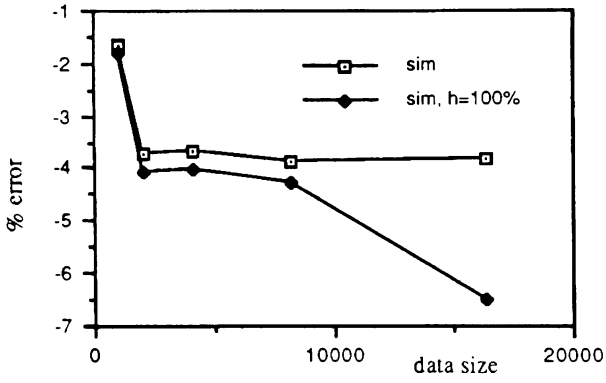

Figure 2.4: % error in timing prediction for the FFT algorithm

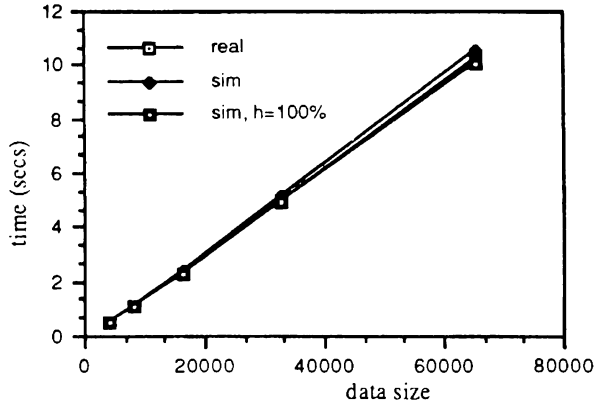

Figure 2.5: Validation of the Sun-3/280 68020 detailed cache simulator using the mergesort algorithm

ratios and shows the effects of varying cache parameters has been developed. Our model is based on the work of Agarwal et al. (1989), but requires fewer parameters and less calculation. It depends on the extraction of a few parameters from the address trace during one detailed execution-driven cache

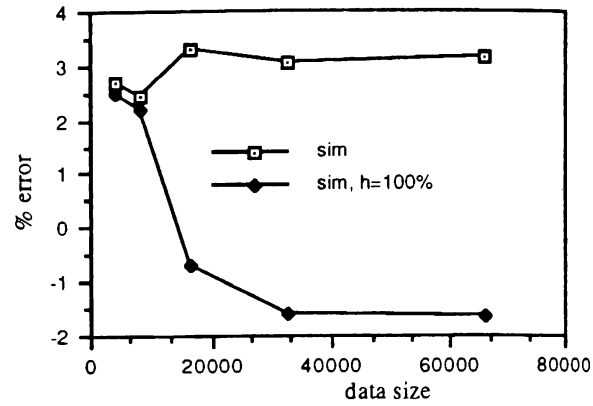

Figure 2.6: % error in timing prediction for the mergesort algorithm

simulation run (described in the previous section), and is the basis of the estimative technique described in the next section.

3.1 The Cache Model

A cache miss may be attributed to any one of a number of effects, as detailed in (Agarwal, et al. 1989). *Start-up effects* correspond to the misses that occur when a process begins execution for the first time on the processor, or when it abruptly changes phases of execution (a phase of execution is a stage in the program with a significantly different working set). *Non-stationary behavior* corresponds to the misses that occur when references are fetched for the first time after the start-up phase. Since the cache has a finite size, multiple memory references of a process may compete for a cache set and collide with each other, causing additional misses due to *intrinsic interference*. An additional source of cache misses is multiprogramming, which causes references by one process to invalidate cache locations that contained valid data for another process (*extrinsic interference*). In a shared memory multiprocessor, the protocol used to maintain coherence also affects the miss ratio, along with other performance parameters, of each individual cache.

Our cache model takes only the start-up, non-stationary, and intrinsic interference effects into account. The underlying assumption is that there is only one active process. The model is separated into two parts — one that is not a function of the cache size, and the other that is a function of the cache size. The variables used in our model are as follows :

- B = line size of the cache in bytes
- S = number of sets in the cache
- D = degree of set associativity of the cache
- U = number of unique addresses referenced by the program
- T = total number of references made by the program
- m = miss ratio for a particular cache organization

The first part of the model characterizes the start-up and non-stationary effects. These effects contribute to the miss ratio of even an infinite cache, since they occur on the first access to a line in main memory, namely, all unique program lines, and can be characterized by the ratio of the number of unique lines in the program to the total number

of references made by the program. The number of unique memory lines accessed by the program is dependent on the distribution of run lengths, their alignment within a cache line, and the distribution of space intervals between runs. A run is defined as the maximum number of sequential or contiguous references made by the program. Agarwal et al. (1989) present a Markov model to characterize run lengths that requires at least three parameters to determine the number of unique lines accessed by the program as a function of the line size. For simplicity, however, the number of unique lines in the program is assumed here to be U/B. The miss ratio that can be attributed to start-up and non-stationary effects is, therefore,

$$U/(B*T)$$

The impact of intrinsic interference depends on how the addresses are distributed over the address space (static), as well as the sequencing of references and the relative frequencies of their access (dynamic effect).

The static characterization of the intrinsic interference effect is based on the assumption that the assignment of program lines to sets in the cache is a random process in which each set is equally likely to be the destination of a line. This assumption appears to be reasonable based on the results by Smith (1987) and Agarwal et al. (1989). Hence, a binomial distribution can be used for the probability that d unique program lines map to a given cache set. Let $P(d)$ be this probability and let u be the number of unique lines accessed by the program, where $u = U/B$. Then

$$P(d) = C(u,d)(1/S)^d(1-1/S)^{(u-d)}$$

Let d_t (the static component of the intrinsic interference effect) be the total number of colliding lines, or the total number of unique lines that map into sets with greater than D lines mapped to them. Then

$$d_t = U/B - S\sum_{d=0}^{D}(dP(d))$$

The dynamic component is characterized by the collision rate, c, which is the average number of collisions per colliding line.

Let

d' = Average number of lines mapped to a set, given that the number mapped is greater than D.

c' = A constant of the collision rate dependent on the program trace characteristics.

S_t = Number of sets that have greater than D lines mapped to them.

Then

$$d' = d_t/S_t$$

The collision rate c is defined as follows :

if $(d' > (D + 1.5 + 0.5log_2 D))$

$$c = c'(d'-D)^{(1/(1+log_2 D))}\, log_2^{(1/log_2(D+1))}(B+1)$$

else

$$c = c'\left(\frac{d'-D-1}{0.5(1+log_2 D)}\right)^{(1+log_2 D)} \times log_2^{(1/log_2(D+1))}(B+1)$$

The expression used to estimate the collision rate is empirical, though some intuitive explanation can be given. The collision rate is inversely proportional to the cache size. This is reflected in its proportionality to $(d'-D)$. It is intuitive that the collision rate is proportional to the average number of colliding lines per set that exceed the set size, given that the sets have a number of lines greater than the set size mapped to them. However, this proportionality to $(d'-D)$ decreases as the set associativity increases. The proportionality of the collision rate to $(d'-D)^{(1/(1+log_2 D))}$ was chosen because of the behavior of all the test programs as the set size increased. There was a sharp increase in hit-ratio as the set size increased from one to two, but a further increase in the set size caused a more gradual change in the hit ratio.

As the line size increases, although the overall miss ratio decreases, the collision rate has a tendency to increase. The use of the proportionality to the log_2 of the line size was suggested by a similar use in Smith's empirical model (Smith, 1987). Smith's model, however, does not account for set size, assuming a fully associative cache. This is because there is a higher probability of replacing a *live* line (a *live* line is one that is likely to be reaccessed in the future (Agarwal, et al. 1989)). This proportionality decreases as the set size is increased in the same way as the proportionality to d'.

Agarwal et al. assume that the collision rate is stable over different cache sizes. They also define the collision rate for d and determine an average collision rate for the entire cache by calculating a weighted mean over all possible values of d. We compute an average value d' for the number of lines that map to any set, and use this to determine the collision rate for the entire cache. The characterization of the collision rate for a given d (or d') is also different.

Thus, the total miss ratio may be represented as

$$miss\ ratio = U/(T*B) + c*d_t/T$$

3.2 Extrapolation of Program Parameters

The program parameters U and T are extracted using one run of the detailed simulation, as is the miss ratio for the simulated cache. These parameters are useful in performance prediction only for program runs that use the same data size. In order to make predictions for the algorithm using a different data size, it is necessary to extrapolate the program parameters.

It is possible to predict the number of unique references for an algorithm given the input data size and the number of unique references for any one other run. It is also possible to predict the total number of references for a particular data size A using parameters extracted from a run for a different data size B for an algorithm that is data-independent (time complexity independent of the *values* of the data), if the time complexity of the algorithm is known. For large data sizes, the change in the number of unique references is proportional

to the change in data size, while the change in the total number of references made by the program depends on the time complexity of the algorithm. The collision rate is a characteristic of the algorithm and should not vary with data size for data-independent algorithms (the manner in which the instruction and data space is accessed is independent of the values of the data or the data size). Thus, the parameters determined from one program run can be used to predict the cache performance not only for different cache organizations, but also for runs using different data sizes.

3.3 Performance Measurements

The performance of the analytical model was determined using the sequential versions of the FFT, SOR (Successive Over-Relaxation), mergesort and quicksort algorithms. In most cases these algorithms were simulated in detail for a cache size of eight Kbytes, set associativity of two, and line size of eight bytes. The parameters extracted from each detailed run were then used to predict the miss ratios for the algorithm for other cache organizations. The size of the cache simulated was chosen so as to hold a fairly large portion of the working set size of the algorithm (at least 20%). This is the normal range of interest, and the range of cache organizations for which the analytical model holds. The data size was chosen so that the programs accessed at least three million addresses. Results from the FFT and quicksort algorithms are presented (see (Dwarkadas, 1989) for additional data).

The graphs in Figure 3.1 show the miss rate versus cache size for the sequential FFT algorithm using a data size of 4096, for set sizes of 1, 2 and 4 and a line size of 8 bytes. The maximum error is 60% while the average error in prediction is 30%.

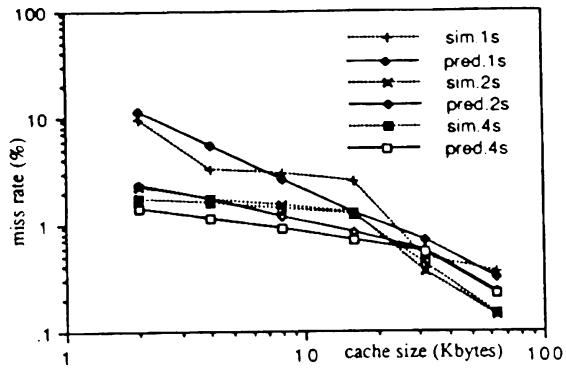

Figure 3.1: Miss rate vs. cache size for FFT - data size=4096 points and line size=8

The miss ratios for the FFT algorithm were also predicted for a data size of 2048. The collision rate used was the same as that predicted for a data size of 4096 and U and T were extrapolated as detailed in the previous section. The results were similar to that for the larger data size, as illustrated by Figure 3.2.

Results for the quicksort algorithm (*qsort*) are shown in Figure 3.3. When the set size is one, the model is not accurate until the cache size is larger than the data size. This is because of successive replacement of data lines still in use. When the set size is larger than one, the predicted miss ratios

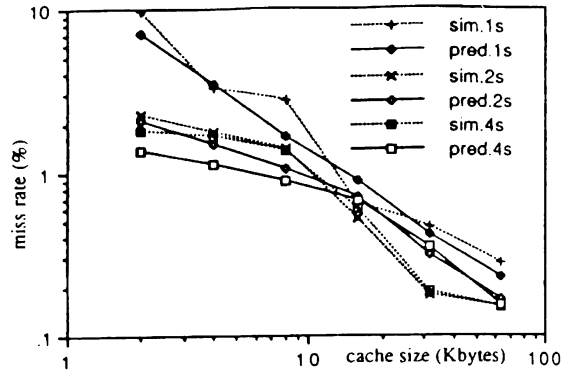

Figure 3.2: Miss rate vs. cache size for FFT - data size=2048 points and line size=8

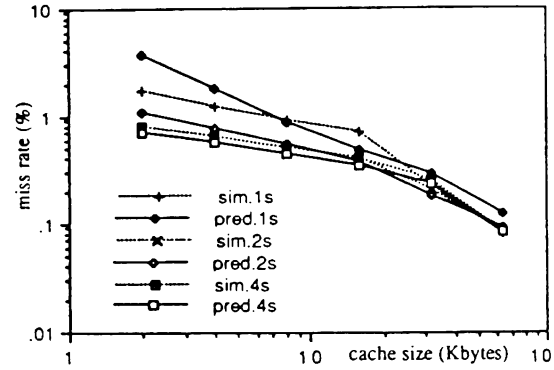

Figure 3.3: Miss rate vs. cache size for qsort - line size=8

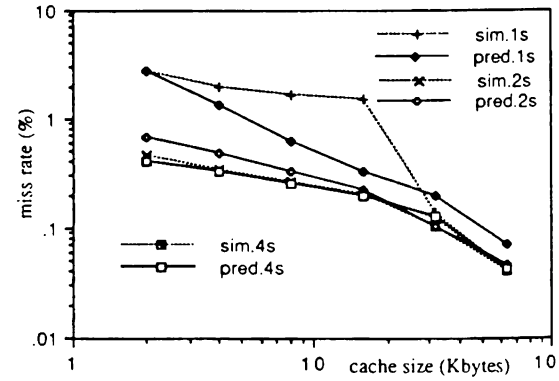

Figure 3.4: Miss rate vs. cache size for qsort - line size=16

show a fairly good correlation, with a maximum error of 45% and an average error of 12%. The data size used here was 10,000 integers.

The predictions made by the model for a set size of one do not always follow the detailed cache simulation results. This is because, for a direct mapped cache, the cache behavior

of the algorithms is far more erratic and sensitive to the actual location of the lines in the virtual address space and the assumptions made by our model about random access patterns no longer hold and the predictions may not be correct. The predictions of the model as the degree of set associativity is increased are fairly accurate.

The performance of the model as the line size is increased is reasonable. The model's predictions, however, do not follow the same trend for certain cache sizes, in which the hit ratio peaks at a particular line size and then begins to fall as the line size is increased further. As the line size is increased, the number of misses due to accesses to unique lines decreases. However, there is a larger probability of a collision and replacement of a line that may be accessed in the future, since each line now contains a larger amount of information. Our model assumes that the collision rate is proportional to the log of the line size. For certain algorithms this is not enough to account for the large collision rates as the line size is increased.

4. AN ESTIMATIVE TECHNIQUE FOR THE SIMULATION OF CACHE-BASED SYSTEMS

Prediction of the cache miss ratio is sufficient to accurately predict the program's execution time, as long as there is no interaction among processes of the program involved. This is not true in the case of several processes synchronizing with each other, either explicitly through message passing or access to shared variables, or implicitly because of access to a common memory. Delays due to cache misses could change the points in time at which processes synchronized, as well as change the order in which synchronization events occurred. The distribution in time of the cache misses becomes equally important for performance prediction.

In a tradeoff between the speed of simulation and the accuracy of its predictions, a profiler that is based on the analytical model presented in the previous section has been designed. The profiler is suitable for use in conjunction with the RPPT and allows an efficient simulation of any uniprocessor or multiprocessor cache-based system.

4.1 Implementation

The profiler is based, as in the case of the detailed simulation, on the concept of basic block profiling (Weinberger, 1984). Instead of generating an address trace, a count of the total number of references made in the basic block and an aggregate time for the basic block (excluding cache misses) are generated.

At the start of the basic block, the profiler inserts code to read the number of references made in the block as well as to determine, as accurately as possible, the time taken to execute the basic block. The overhead due to cache misses is calculated by assuming a constant overhead per miss, which can be determined after considering factors such as bus delay and contention (in the case of a common main memory). The aggregate miss ratio generated by the analytical model using parameters from a detailed simulation run is applied to the total references for the block to estimate the number of cache misses in the block.

4.2 Performance Measurements

We compared the time predicted by the estimative technique with the time predicted by a detailed simulation for several parallel algorithms. The simulated system was a shared-bus multiprocessor with explicit message passing. The processors have independent private caches and common main memory, the caches communicating with main memory by means of another shared bus. No sharing of writable data is allowed. The cache organization simulated was a write-through write-allocate cache with a least recently used replacement policy.

Figure 4.1 presents the results for the eigenvalue/ eigenvector algorithm for 1, 2, 4, and 8 processors, showing the timings obtained from the detailed simulation (*det.*p*, where '*' is the number of processors being simulated) and the simpler estimative simulation (*simple.*p*). The two curves follow the same general trend. The data size used was a 16×16 matrix. The error in timing prediction can be seen to be within 1% for caches larger than 8 Kbytes (Figure 4.2), and within 3% of the results from the detailed simulation for caches as small as 2 Kbytes.

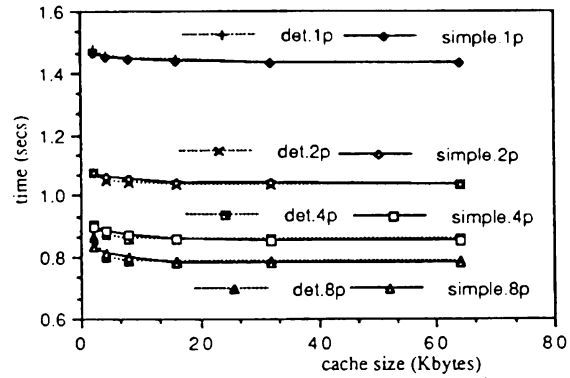

Figure 4.1: Comparison of detailed and estimative simulation timings - eigenvalue, line size=8, set size=2

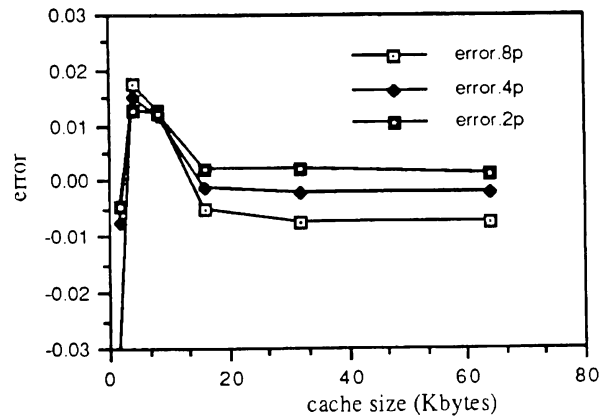

Figure 4.2: Error in prediction by the estimative simulation for the eigenvalue algorithm

The accuracy of the estimative simulation is heavily dependent on the accuracy of the analytical model used. Additional sources of error are introduced by the use of a

uniform hit ratio through the execution of the entire algorithm, a situation that is far from reality. This can change the time at which messages are sent and received, and thus affect the overall execution time. A sequence of cache miss ratios might be used to give better accuracy. This can be done by splitting the program into phases, possibly at synchronization points or at points where there is a significant change in the program's execution behavior, or using intervals with a convenient and large number of references, depending on the desired accuracy. This involves a tradeoff between speed and accuracy.

5. OVERHEAD COMPARISON

Figure 5.1 shows the overhead of the different simulation detail levels as compared to the time for actual execution of the program in the case of a uniprocessor. The overhead added to the numerical algorithms, SOR (*sor*) and FFT (*fft*), can be seen to be much less than that added to the non-numerical algorithms since they involve floating point, and hence do more computation between simulation points. Hence, the overhead for the estimative model (*estimative simulation*) in the case of the non-numerical algorithms is in the range of 18 while for the numerical algorithms, it is in the range of 4.5. Speedup of the estimative simulation as compared to the detailed simulation is in the range of 12–15. Two data sizes were used for the FFT algorithm, 2048 (*fft11*) and 4096 (*fft12*), in order to show that the overhead is constant for a given algorithm and does not vary with the size of the computation.

The overhead added by the detailed execution-driven simulation for trace generation alone (*trace generation*) is in the range of 15–48, the low end being for the numerical algorithms that use floating point. Simulation of a direct-mapped cache (*det. dmap cache sim.*) doubles this overhead (35–90), while simulation of a full-fledged set-associative cache (*det. set-assoc. cache sim.*) increases the overhead by a factor of six (100–325). Hence, the total overhead can be largely attributed to processing the address trace. The overhead associated with I/O activity is completely removed.

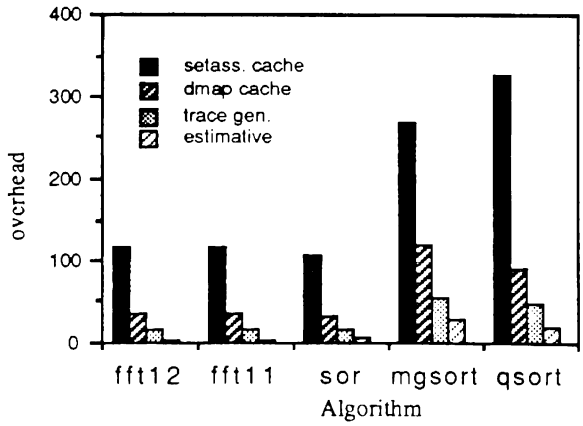

Figure 5.1: Overhead comparison for different levels of simulation detail

Simulations of parallel algorithms on a parallel architecture were also carried out. Results for the two-processor case are shown in Figure 5.2. The overhead for each multiprocessor simulation is determined as the ratio of the time it takes to execute to the time taken to carry out an RPPT simulation without emulation of the cache. The algorithms used were the eigenvalue/eigenvector (*eigen*) determination and the mergesort (*mgsort*), and the simulated architecture was that described in the previous section.

Simulation of the cache bus added a fairly large overhead to the total execution time of the simulation in the case of a write-through cache (*det. sim., WTWA*), since it required process synchronization at every write and at every miss. The overhead with (*det. sim., WB*) and without (*det. sim., nobus*) modeling the cache bus is shown in the case of a write-back algorithm as well. These overheads are more than double that for simulating a direct-mapped cache (*det. sim., dmap*). The overhead of the estimative simulation (*simp. sim.*) is also shown, and is a factor of 2 for the eigenvalue algorithm, while the overhead is 8 for the mergesort algorithm. The simulation overhead incurred merely to generate the address traces (without writing them to a file) is also shown (*det. sim., trace gen*).

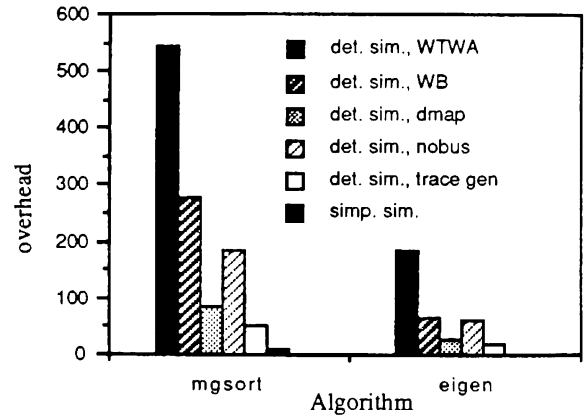

Figure 5.2: Overhead comparisons for different levels of multiprocessor simulation detail (2 processors)

The RPPT simulations without emulation of cache behavior have been found to add an average overhead in the range of 1.5 to 15 (Covington, et al. 1989) to the Concurrent C (Madala, 1987) versions of the programs. In the case of the mergesort and eigenvalue algorithms, this overhead is around 2.

6. CONCLUSIONS

We have developed a software execution-driven method for generating address traces and simulating caches, providing results almost as accurate as conventional instruction-level simulation but with higher efficiency. We have also developed an estimative technique for determining cache and system (uniprocessor and multiprocessor) performance, as a tradeoff between speed and accuracy.

In order to simulate shared memory multiprocessors with private caches and shared, cacheable data, the RPPT's support of shared memory systems must be extended, and the design of the cache simulator must also be extended to support cache coherency mechanisms. These extensions are currently being implemented, and will enable the generation of parallel traces for shared memory multiprocessors, which are a scarcity at present. The RPPT could also be used to

simulate the effects of multiprogramming in a single processor, and to determine the effects of varying the time slice on the performance of its cache. Thus, these tools in conjunction with the RPPT provide a powerful environment for the study of cache-based systems.

REFERENCES

Agarwal, A., 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*, pp. 119–127, June 1986, vol. 14, no. 3.

Agarwal, A., M. Horowitz, and J. Hennessy, An Analytical Cache Model, *ACM Transactions on Computer Systems, 7*, pp. 184–215, May 1989.

Clark, D. W., Cache Performance in the VAX-11/780, *ACM Transactions on Computer Systems, 1*, pp. 24–37, February 1983.

Covington, R. G., *Validation of Rice Parallel Processing Testbed Applications*, Ph.D. thesis, Rice University, Houston TX, December 1988.

Covington, R. G., 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*, pp. 4–11, Santa Fe, NM, May 1988.

Covington, R. G., S. Dwarkadas, J. R. Jump, G. Lauderdale, S. Madala, and J. B. Sinclair, *The Efficient Simulation of Parallel Computer Systems*, TR 8904, Department of Electrical and Computer Engineering, Rice University, Houston, TX, March 1989.

Dwarkadas, S., *Efficient Methods for Cache Performance Prediction*, Master's thesis, Rice University, Houston TX, May 1989.

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

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

Smith, A. J., Cache Memories, *Computing Surveys, 14*, September 1982.

Smith, A. J., Line (Block) Size Choice for CPU Cache Memories, *IEEE Transactions On Computers, C-36*, September 1987.

Thiebaut, D. F. and H. S. Stone, Footprints in the Cache, *ACM Transactions on Computer Systems, 5*, November 1987.

Weinberger, J. P., Cheap Dynamic Instruction Counting, *AT&T Bell Laboratories Technical Journal, 63*,, October 1984.

AUTHORS' BIOGRAPHIES

SANDHYA DWARKADAS is a Ph.D. student in the Department of Electrical and Computer Engineering at Rice University. She received her B.Tech. degree in Electronic Engineering from the Indian Institute of Technology, Madras (India) in 1986, and M.S. degree in Computer Engineering from Rice University in 1989. Her current research interests include parallel computer architecture, concurrent programming, distributed processing, and simulation methodology.

Sandhya Dwarkadas
Department of Electrical and Computer Engineering
Rice University
Houston, TX 77251–1892
(713) 527–8101–3598

J. ROBERT JUMP received his B.S. and M.S. degrees in Electrical Engineering from the University of Cincinnati in 1960 and 1962 respectively. He also received the M.S. and Ph.D. degrees in Computer and Communications Science from the University of Michigan in 1966 and 1968 respectively. He is currently a professor in the Department of Electrical and Computer Engineering at Rice University in Houston, TX. His current research interests include several aspects of parallel computing systems such as interconnection networks, system organization, and the simulation of parallel computers. Dr. Jump is a member of the ACM and IEEE, and has served as an editor of the *IEEE Transactions on Computers*.

J. Robert Jump
Department of Electrical and Computer Engineering
Rice University
Houston, TX 77251–1892
(713) 527–8101–3576

JAMES B. SINCLAIR (M'78) received the Ph.D. degree from Rice University in 1978. After receiving his master's degree from Rice in 1974, he worked for a time as a VLSI design engineer with Texas Instruments in Stafford, Tx. Currently he is an Associate Professor in the Department of Electrical and Computer Engineering at Rice University. He is a member of the IEEE Computer Society, ACM, Tau Beta Pi, and Sigma Tau. His research interests include distributed processing, computer networks, parallel computer architecture and concurrent programming, simulation methodology, and performance evaluation.

James B. Sinclair
Department of Electrical and Computer Engineering
Rice University
Houston, TX 77251–1892
(713) 527–8101–3596