# The Time-Space Model for Instruction Reference Behavior

Che-Chi Weng and Eric E. Johnson
Parallel Architecture Research Lab
New Mexico State University
ejohnson@nmsu.edu

## ABSTRACT

In this paper, an abstract model is established for describing the behavior of instruction references. From this model, we can obtain accurate predictions of cache miss ratios without the expense of trace-driven simulation.

The time-space model explores the relationship between the sojourn times and working spaces of program executions. The average sojourn time per reference (ASPR) of each block of program activity is used as an indicator of the likelihood that such blocks stay in a cache. A time-space list is created, ordered by ASPR, to keep track of how much time is spent in each block of the working space. This list incorporates the information as to phase separations, loop sizes, loop counts, and so on.

With this time-space relation of a trace and a few constants regarding the cache memory configuration, the miss ratio curve of a program can be predicted, based on the assumption that the higher ASPR blocks always stay in the cache.

We present an evaluation of this technique using a wide variety of programs. The miss ratio curves predicted by the time-space model for fully associative caches are compared with the curves produced by trace-driven simulation, with very good agreement in most cases.

## INTRODUCTION

The study of program behavior was brought into focus in the mid-1960s, when the responsibilities of memory management were transferred from programmers to operating systems due to the wide use of virtual memory and multiprogramming techniques. The purpose of the program model was to provide a basis for efficient use of system resources.

As the speed of processing units increased, memory speeds could not keep up. Therefore, memory access became a bottleneck to the system performance. The use of cache memory, a small high-speed memory to store items that are recently used and likely to be used in the future, can reduce the average memory access time. However, the performance of cache memory depends strongly on its design (i.e., its configuration and replacement algorithm), as well as the memory reference patterns in application and system programs.

Because no globally optimal design parameters for cache memories are available, researchers have sought to develop analytical models of program behavior. The goal of these models is to accurately predict cache performance without the need to actually build each cache design and measure its performance under real workloads. Because cache performance is sensitive to the dynamic behavior of a program, these models are parameterized using measurements of programs of interest, often derived from traces of executions of those programs.

Several models for program behavior have been studied in the last three decades. Many of the earlier models were developed for study of virtual memory paging, but may also be applied to cache designs. The Random Reference model (RRM) [1] assumes that each page (or cache line) is equally likely to be referenced at any given time. The number of pages in a program is the only parameter it uses.

The Independent Reference model (IRM) [2] assigns a fixed stationary probability to each page of being referenced at any given time. Thus, for a program with N pages, there are N-1 independent parameters.

The approach of the Stack model [2], also known as the Simple LRU Stack model, is similar to that of the IRM. However, instead of collecting the probability of each address of being referenced, the Stack model collects the probability of each location in the stack being referenced. In other words, it collects the stack distance probabilities.

Thiebaut's model [3] explores both the locality and the working set size of a program. The size of program space is introduced as well. However, a few important characteristics with great impact on the miss ratios are neglected, such as loop sizes, loop counts, and dead code within a program.

These models represent increasingly sophisticated models of program behavior. However, the miss ratio predictions made by even the best of these models often varies widely from the miss ratios from trace-driven simulation. Therefore, designers must currently resort to such time- and storage space-consuming simulation when accurate miss ratio predictions are needed.

We developed the Time-Space Model in an attempt to more accurately model the detailed memory access behavior of instruction fetching by real programs. A related model of data reference patterns is also being developed. Our motivation for studying instruction and data reference behaviors separately comes partly from the simplicity resulting from separate models, and partly from the popularity of separating data and instruction caches in CPU implementations.

## THE TIME-SPACE MODEL

The time-space model is the simplest of three models of instruction reference behavior that use data collected from the execution of programs of interest [6]. The model is based on a

This work was supported in part by the U.S. Army Research Office under grant DAAH04–93-0229.

summary of instruction reference activity of a program in a list that ranks blocks of the program in order of frequency of execution. This list is then used to directly plot the predicted miss ratios of caches of varying sizes, without resort to simulation.

**Program Blocks**

A *program block* (or simply a *block*) in the time-space model is defined to be a piece of contiguous code such that each instruction in the block has a similar reference pattern, both in reference density and in interarrival time. A block includes one or more contiguous cache lines that will be simultaneously active.

In this model, a block consists of sequential references and *small* forward jumps. (Thus, the blocks in this model do not always correspond one-to-one to compiler basic blocks.) A backward jump, whatever its displacement, is treated as a jump to the beginning of some block, either to itself or another block. Since a big forward jump may jump over a large piece of dead code (code that lies within the address space but is not referenced), it must also be considered as a jump to another block.

In choosing a displacement threshold to distinguish between intra- and inter-block forward jumps, we must consider the effect of including areas of dead code within a block. A piece of dead code larger than the cache line to be simulated should not be contained in a block, or else we may overestimate the miss ratio. However, a piece of dead code smaller than the line size should be contained in a block. Otherwise, the miss ratio can be underestimated. In general, the threshold displacement should be larger than the cache line size, and less than twice the line size. In the results reported later in this paper, a threshold of 1.5 times the line size was chosen.

Thus, any backward jump or big forward jump is considered to be a jump to the beginning of a block, and the instruction that produces a backward jump or a big forward jump is therefore the end of that block.

**Information Collection**

The record of execution (e.g., a trace) of a program is used to partition the executable code of the program into blocks. The following characteristics are collected for each block:

a) its size,

b) the number of visits to a block (V) recorded during a trace,

c) the average number of repetitions of a block per visit ($R_v$),

d) the total sojourn time of a block,

e) the average sojourn time per reference (ASPR) of a block, and

f) the virtual time (in number of references) of the first and last visit to a block.

The number of visits to a block is identified by counting the number of jumps to the block from *other* blocks.

The average number of repetitions of a block per visit is obtained by dividing the number of times the first address of a block is accessed by the number of visits to that block. A block with $R_v$ greater than one is a *loop block* and the $R_v$ value is the average loop count per visit to that block.

Sojourn time is measured in units of instruction fetches. The total sojourn time of a block is the summation of the sojourn time of each visit (including all loop iterations) over all visits.

The ASPR is the total sojourn time of a block divided by the average number of references in that block. The ASPR is a good indicator of how intensively a block is referenced as well as how great an influence the block has on the overall behavior of the program. Therefore, in this model, a higher ASPR block is assumed to occupy the cache memory with higher priority.

An important characteristic of the behavior of a program is its loop sizes and the loop counts. Program loops can have strong positive or negative effects on cache miss ratios. When the size of a loop block is smaller than the cache size, the block may always stay in the cache. The higher the loop count, the lower the miss ratio. However, if the size of a loop block is larger than the cache size, its execution results in cache thrashing. In this case, looping does not increase the hit rate, but decreases it: the higher the loop count, the greater the fraction of all references the block represents, and the higher the miss ratio.

Another important characteristic of the behavior of a program is its phase separations. A program may reference one part of its code during one phase and visit another part of the code during another phase, with both the space and the time disjoint. This behavior greatly reduces the average working set size of a program [4.1] and, hence, the miss ratio. Therefore, the information concerning phases also needs to be captured.

**Time-Space Lists**

A time-space list is a table that shows how much of the total execution time is spent on how much of the working space. It is used in our model to predict the miss ratio curve of a program. Table 1 shows an example time-space list.

Because blocks with higher ASPR are more likely to occupy the cache, they are processed first. A time-space list is created from the block data for all phases concurrently as follows:

a) Select the block with the highest ASPR of those blocks not yet processed.

b) If a block has an $R_v$ value (mean loop count) greater than one and its size is smaller than the largest accumulated size for any phase (the current maximum working space), hold this block in a waiting list. Add its sojourn time to a global sojourn time counter, but defer adding its size to the working space of any phase.

c) Otherwise, if the block does not overlap (in time) with the blocks already processed, create a new phase record. Record the size of the block and the time of the first and the last visit to that block as the size and duration of that phase. Add the sojourn time of the block to the global sojourn time counter.

d) If the current block has an overlap in time with any block(s) already considered (in any phase), add the size of the new block to the record of that phase and update the time of the first and last visits for that phase, merging phases as appropriate. Add the sojourn time of the current block to the global sojourn time counter.

e) If blocks in the waiting list have numbers of visits (V) greater than the current ASPR value, remove each from the waiting list and add it to the appropriate phase according to the times of the first and last references to that block.

f) As the final step in processing each new block, create a new entry in the time-space list. Use the largest accumulated size among all phases as the working space of that entry. Use the current value of the global sojourn time counter as the accumulated sojourn time of that entry (shown in Table 1 as a percentage of total sojourn time).

Table 1: Time-space list of the `tex` trace

| *ASPR* | *space* | *%soj* |
|---|---|---|
| 29844.0 | 20 | 0.262423 |
| 7465.0 | 32 | 0.301808 |
| 7465.0 | 40 | 0.328064 |
| 7465.0 | 48 | 0.354320 |
| 7464.0 | 116 | 0.577469 |
| 7464.0 | 124 | 0.603722 |
| 7463.0 | 164 | 0.734969 |
| 7463.0 | 188 | 0.813717 |
| 7462.2 | 212 | 0.892456 |
| 3732.0 | 232 | 0.925272 |
| 3732.0 | 252 | 0.958088 |
| 3732.0 | 276 | 0.997468 |
| 295.2 | 292 | 0.999545 |
| 3.0 | 308 | 0.999566 |
| 3.0 | 364 | 0.999639 |
| 3.0 | 388 | 0.999671 |
| 3.0 | 412 | 0.999703 |
| 2.5 | 460 | 0.999756 |
| 2.4 | 516 | 0.999814 |
| 2.2 | 548 | 0.999845 |
| 2.1 | 676 | 0.999961 |
| 1.9 | 720 | 0.999998 |
| 1.0 | 724 | 1.000000 |

The ASPR column shows the ASPR of each block in the order examined, space column gives the largest accumulated space among phases (working space), and the %soj column is the percentage of total sojourn time, or the percentage of the length of a trace, spent in a working space of that size.

Note that ASPR can be less than one if a block contains bubbles and is visited only a few times. The last entry of the space column tells the largest accumulated space among phases for the whole trace; it is not necessarily the summation of the size of all blocks in the trace, because one part of the address space of a trace may never overlap in time with other parts, and the working space records only the phase with the largest accumulated size. The last entry of %soj column should always be one.

**Prediction of Miss Ratio Curve**

To estimate the miss ratio curve, we must consider how a miss can occur. According to [5], a miss can be categorized as one of the following:

a) *Capacity Miss* -- The capacity miss is caused by insufficient cache size. It happens when the working set size is larger than the size of cache memory, because potentially useful cache lines must be evicted from cache to make way for the newly referenced lines.

b) *Conflict Miss* -- The conflict miss is due to contention for the same cache line among lines mapping into the same set.

c) *Cold Start Miss* -- The cold start miss is result from the first time reference to a cache line.

A few additional terms that we use in this context need to be defined:

a) *Static utilization of a cache line $U_s$*: the ratio of the average number of references that fit into a cache line to the number of reference slots available in a cache line, or.

$$U_s = \frac{\text{number of unique instruction references in the trace} \times \text{instruction size}}{\text{number of cache lines needed to store the trace} \times \text{line size}}$$

Both the instruction and line size are measured in bytes.

b) *Dynamic utilization of a cache line $U_d$*: the ratio of the average number of continuous references to a cache line to the number of reference slots in a cache line, or.

$$U_d = \frac{\text{number of instruction references in the line used} \times \text{instruction size}}{\text{line size}}$$

c) *Effective program space of a trace EPS*: the size of a cache actually needed to store the whole trace, derived by summing up the sizes of all blocks including the extra margin needed for incomplete lines. Thus, the EPS includes bubbles (dead code) within a block as well as bubbles of incomplete lines at the margins of blocks, but excludes large areas of dead code between blocks.

EPS is the threshold cache size beyond which a cache behaves as an infinite cache for the measured program. The RRM, IRM, stack, and Thiebaut's model collect similar information by counting unique pages or references in a trace. This approach implies that the static utilization of a cache line is one, which rarely happens. As a result, these models always underestimate the miss ratio at large cache sizes. By using the EPS approach, the time-space model always gives a very accurate miss ratio prediction at large cache sizes.

Prediction of the miss ratio curve is based on the following observations and assumptions:

a) At large cache sizes (EPS less than the size of cache), no misses will occur after the cold start misses. The cold start misses can easily be calculated by using the EPS.

b) The miss ratio at the smallest cache size, exactly one cache line, can be derived from the dynamic utilization of a trace.

In our approach, however, we assume it to be a constant that depends only on the size of cache line.

c) At medium cache sizes, when the cache size is less than the EPS, we make the rather bold assumption that blocks with working space less than the size of the cache always stay in the cache, and that the capacity misses of the blocks that we ignore are equal in number to the capacity misses from other blocks that we overestimate. Thus, for a cache of size Z, the time spent on a working space z (the largest working space that is contained in Z) incurs no cache misses. The remaining time, 1-%soj for that working space, is all cache misses. This assumption is the basis of the time-space model.

A typical miss ratio curve, plotted in double logarithmic scale, is shown in Figure 1. This serves as a tool for explaining our prediction of a miss ratio curve. The parameters of the time-space model are listed in Table 2.

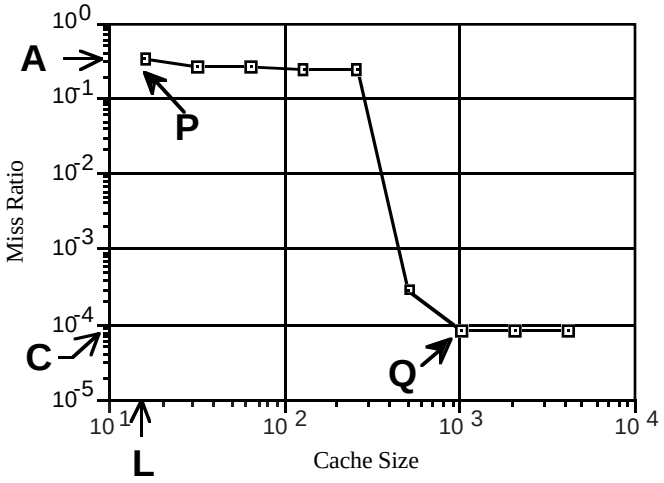


Figure 1: Miss Ratio Curve of the `tex` Trace

Table 2: Parameters of Time-Space Model

| *Parameter* | *Definition* |
|---|---|
| T | the length of the trace, in references. |
| $B_i$ | the size of block i, in bytes. |
| N | the number of blocks. |
| L | the line size of the cache, in bytes. |
| I | the instruction size, in bytes. |
| S | the accumulated program space, in bytes (sum of all $B_i$) |
| EPS | effective program space (see text) |
| $R_r$ | the program space reduction ratio: $R_r = S / EPS$ |

Since cache lines at both margins of a block often contain unused code, as mentioned earlier, simply adding up the sizes of all blocks underestimates the number of cache lines actually needed to store the program space. The reduction ratio is a function of average block size and the dead code between blocks. The smaller the average block size the more the number of cache lines is underestimated. Less dead code between blocks also reduces the estimate. From our observations, the reduction ratio is about .93 for caches with 16-byte lines, 0.85 for 64 byte lines, and 0.7 for 256-byte lines.

The miss ratio curve is predicted by the following algorithm:

a) *Determine A, the maximum miss ratio for the cache*. A is a function of the dynamic utilization of the cache line of a trace. It can be as small as 1/(L/I), and can be as large as 1. From our observations, it is about 0.33 for a cache with 16-byte line size, 0.125 for a cache with 64-byte line size, and 0.068 for a cache with 256-byte line size.

b) *Determine C the minimum miss ratio.* C = (EPS/L)/T.

c) *The miss ratio curve between point P and point Q* can be estimated as follows

i) For cache size K, look in the space column in the time-space list for the largest value that is smaller than ($R_r$*K), where $R_r$ is the space reduction ratio defined above.

ii) use the %soj of that row to calculate the miss ratio (m) for cache size K as m = A * ( 1- %soj )

iii) if m < C, then m = C.

## RESULTS AND COMPARISONS

The time-space model has been evaluated by using the algorithm described above to predict the miss ratio curves of several traces, and comparing the results with those of the real traces. The traces used represent a wide variety of program workloads. They are generated from "real" application programs, mostly SPEC 92, with tens of millions of instruction references. A summary of the traces is given in the Appendix.

In the evaluations, we observed three distinct types of program behavior according to the shape of miss ratio curves:

a) smoothly descending curves,

b) curves containing mainly big drops and horizontal lines, and

c) curves with a combination of the above.

In the following, one trace was selected from each type for evaluation at 16-, 64- and 256-byte line sizes. Since the time-space model does not take into consideration conflict misses, the predicted miss ratio is compared with the fully-associative cache miss ratio of a real trace. The results and comparisons are shown in Figure 2 through Figure 10.

Figures 2 through 4 show the miss ratio curves of the `mdljdp2` trace and our prediction using the time-space model. In Figure 2 (16-byte line), the miss ratio curve between 16-byte and 512-byte cache sizes is almost flat. This is because a 564-byte block dominates the trace (93.42% of the sojourn time). Therefore a cache smaller than 564 bytes will thrash. This trace spent 99.773% of its sojourn time in a 876-byte heavily executed working space. Thus, we can expect a big drop at 1K-byte cache size. If we subtract the fraction of the time it spends within the 876-byte space from 1.0, we have 0.00227. This is exactly the depth of the miss ratio drop from a 512-byte cache to a 1K-byte cache. The agreement between predicted and actual

miss ratios is excellent over the five orders of magnitude in cache sizes and four orders of magnitude in miss ratios for this program, and for most other programs evaluated.

For the `alvinn` and `cexp` traces, Figures 5 through 10, we noticed that the predictions are slightly shifted either to the left or to the right of the real miss ratio curves. This is because the reduction ratio $R_r$ was assumed to be a constant; for a trace with higher actual $R_r$, our prediction is likely to shift to the right of the real curve (overestimating the miss ratio), and vice versa.

From these figures, we notice that the predictions of miss ratios on small cache sizes are always fairly close to the trace-driven simulation results. This shows that the approach of using a constant to predict the miss ratio at small cache size is not only simple, but effective. At large cache sizes, the miss ratios are perfectly matched for all traces. Thus, the EPS approach for estimating program space appears excellent. For intermediate cache size, although we do not get a perfect match for some programs, the shapes of the predictions are always quite similar to those produced by simulation.

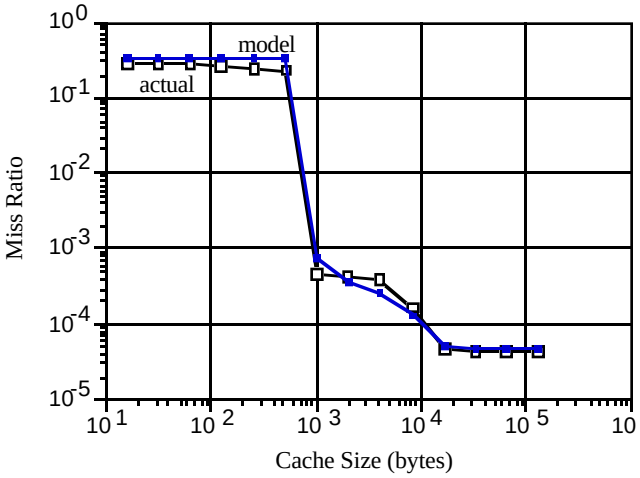


Figure 2: Miss ratio curves of the mdljdp2 trace (L = 16)

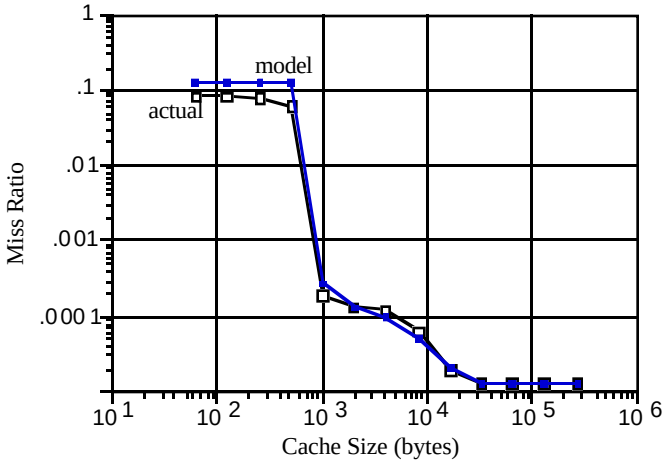


Figure 3: Miss ratio curves of the mdljdp2 trace (L = 64)

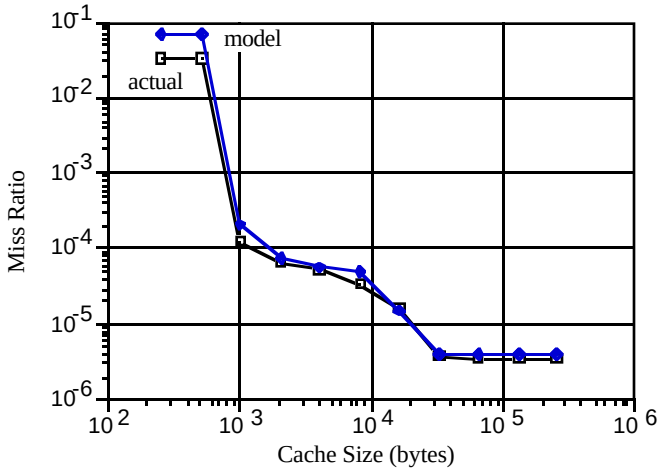


Figure 4: Miss ratio curves of the mdljdp2 (L = 256)

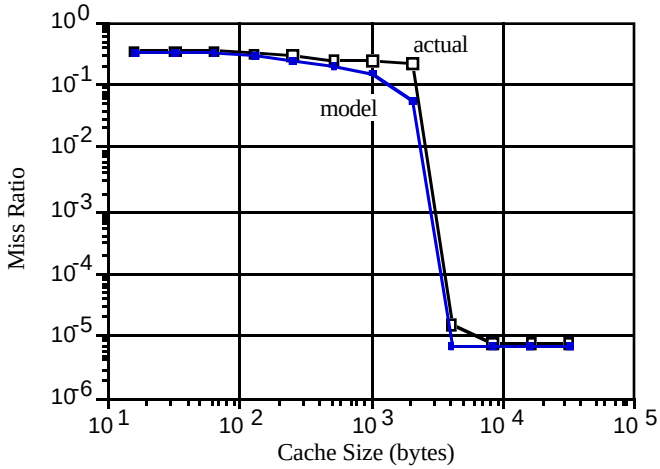


Figure 5: Miss ratio curves of the alvinn trace (L = 16)

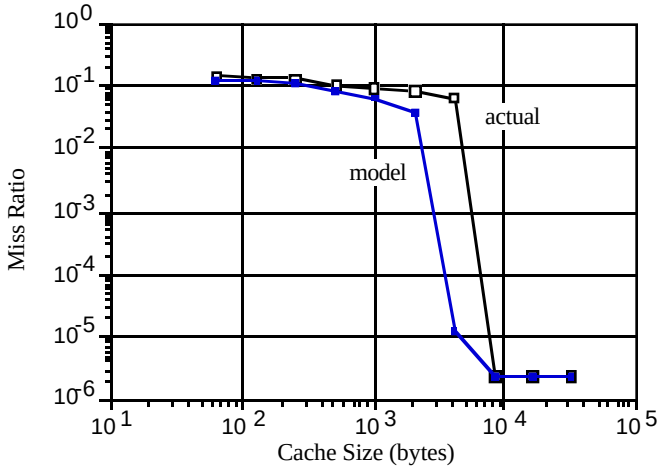


Figure 6: Miss ratio curves of the alvinn trace (L = 64)

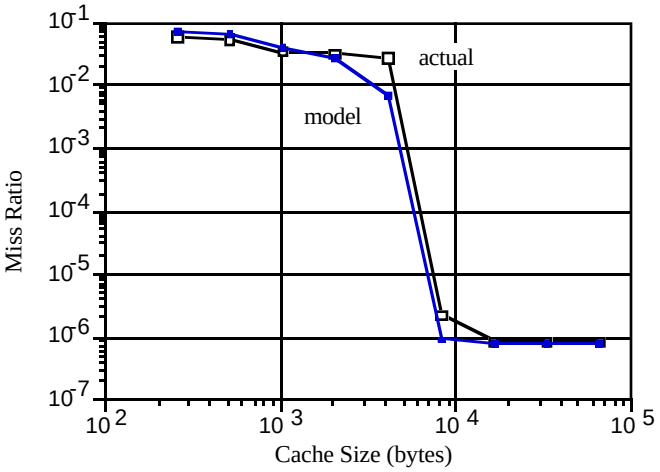


Figure 7: Miss ratio curves of the alvinn trace (L = 256)

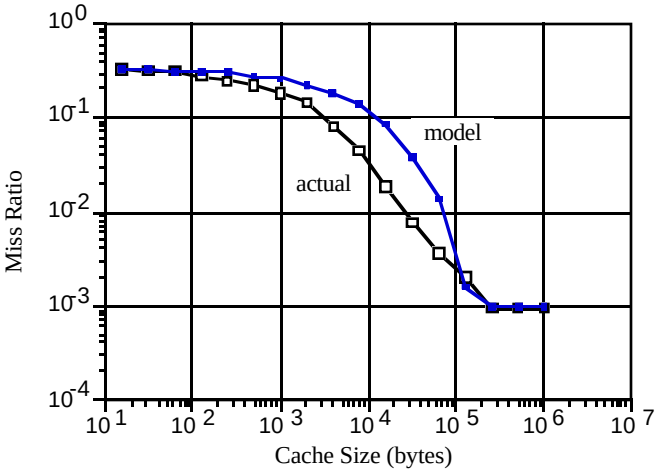


Figure 8: Miss ratio curves of the cexp trace (L = 16)

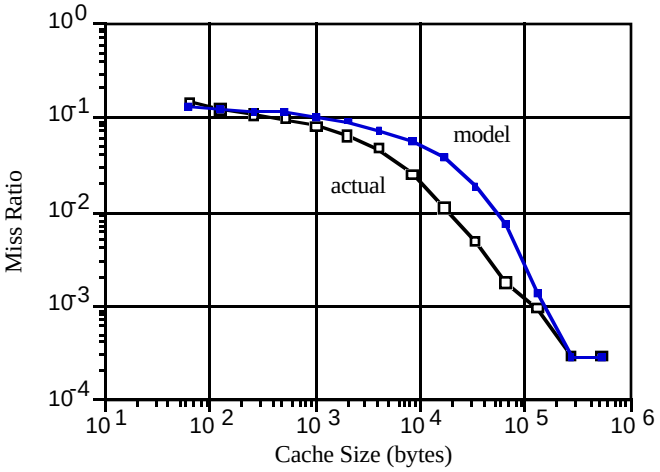


Figure 9: Miss ratio curves of the cexp trace (L = 64)

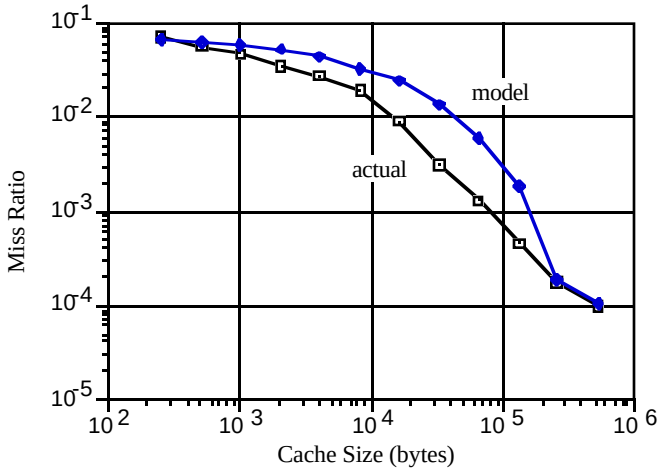


Figure 10: Miss ratio curves of the cexp trace (L = 256)

## CONCLUSIONS

In this paper, we introduced the time-space model for instruction reference behavior, which accurately predicts instruction cache miss ratios without simulation. The technique employs a list of block sizes and execution statistics collected from execution of a program, and predicts miss ratios based on the assumption that the intensively executed blocks of code always stay in the cache. We conclude that the cache-relevant behavior of a program is dominated by the effective program space, intensively referenced blocks, loop sizes, loop counts, and, most importantly, the time-space relationships among block executions.

Miss ratio curves can also be drawn directly from the time-space list of a trace by plotting either 1- %soj versus working space or ASPR versus working space, with some scaling [6].

To date, we have derived the time-space lists used to predict cache miss ratios from traces. We are currently investigating techniques to produce the time-space lists instead from program load maps and standard profiling utilities, and to extend the applicability of the technique to accommodate conflict misses. If successful, this may permit architects to dispense with trace-driven simulation entirely in the design of instruction caches.

## REFERENCES


1. H. Opderbeck and W.W. Chu, "The Renewal Model for Program Behavior", *SIAM J. Comput.* 4(3) Sept 1975.
2. J.R. Spirn, *Program Behavior: Models and Measurements*, Elsevier Computer Science Library, 1977.
3. D. Thiebaut, J.L. Wolf, and H. S. Stone, "Synthetic Traces for Trace-Driven Simulation of Cache Memories", *IEEE Transactions on Computers* 41(4) April 1992.
4. A.J. Smith, “Cache Memories,” ACM *Computing Surveys* 14(3) September 1982.
5. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1990.
6. C.C. Weng, "Models for Instruction Reference Behavior," Ph.D. dissertation, NMSU, 1993.

## APPENDIX: TRACES USED IN EVALUATION

| *Name* | *Lang* | *Precision* | *CPU* | *Refs* | *Description* |
|---|---|---|---|---|---|
| mdljdp2 | fortran | double | R2000 | 65M | solves the equations of motion for a model of 500 atoms interacting through the idealized Lennard-Jones potential. |
| mdljsp2 | fortran | single | R2000 | 65M | single-precision version of mdljdp2. |
| wave5 | fortran | single | R2000 | 65M | solves Maxwell's equations and particle equations of motion. |
| swm256 | fortran | single | R2000 | 65M | solves a system of shallow water equations using finite difference approximations on a 256*256 grid. |
| awk2 | C | integer | R2000 | 63M | Unix pattern scanning and processing application. |
| alvinn | C | single | R2000 | 59M | trains a neural network called ALVINN using back-propagation to keep a vehicle from driving off of a road. The neural net has 1220 input units, 30 hidden units, and 32 output units, and is fully connected. |
| cexp | C | single | R2000 | 14M | from Gnu C compiler. |
| comp | C | integer | R2000 | 8M | uses Lempel-Ziv coding for data compression. Compresses a 1 MB file 20 times. |
| ucomp | C | integer | R2000 | 5.6M | the uncompress version of comp. |
| sed | C | integer | R2000 | 7.7M | Unix batch editor. |
| cc1 | C | integer | DLX | 0.8M | the GNU C compiler for 68000, version 1.26. |
| spice | fortran | double | DLX | 0.8M | a computer-aided circuit analysis tool. |
| tex | C | integer | DLX | 0.6M | text-processing software for document formatting. |

The R2000 traces were provided by Nadeem Malik of IBM. The DLX traces were provided through Morgan-Kaufmann Publishers. Send e-mail to tracebase@nmsu.edu for information on the availability of these and other traces.