Miss Rate Prediction across All Program Inputs

Yutao Zhong, Steven G. Dropsho, and Chen Ding

Computer Science Department
University of Rochester
{*ytzhong,dropsho,cding*}*@cs.rochester.edu*

Abstract

Improving cache performance requires understanding cache behavior. However, measuring cache performance for one or two data input sets provides little insight into how cache behavior varies across all *data input sets. This paper uses our recently published locality analysis to generate a parameterized model of program cache behavior. Given a cache size and associativity, this model predicts the miss rate for arbitrary data input set sizes. This model also identifies critical data input sizes where cache behavior exhibits marked changes. Experiments show this technique is within 2% of the hit rate for set associative caches on a set of integer and floating-point programs.*

1 Introduction

The efficiency of a computer system to execute a program depends highly on the effectiveness of the memory hierarchy to supply requested data quickly. However, since the structure of the processor's memory hierarchy is almost always fixed, the overall efficiency depends on how well the program reference pattern matches the given cache structure. The memory reference pattern of the program itself depends on the particular data set used for input. Thus, it is necessary to understand the fundamental memory reference behavior of a program *across all input data sets* to determine how effective the memory hierarchy will be at efficiently supporting the program in general.

The effectiveness of memory cache hierarchies depends on the locality of data accesses in the programs. Measurements of locality are made in one of three primary ways. One method is to have compilers perform sophisticated analyses of loop nests to estimate memory access patterns statically. A second method employs profilers to run an application with one or more input data sets and directly measure the dynamic memory behavior, but often with significant performance overhead. The third method uses analysis at run-time to *infer* behavior by subsampling activity. Limited sampling keeps the measurement overhead to a minimum. The latter two methods, profiling and run-time analysis, reveal behavior relative to a particular input data set. Static compiler analysis can provide behavior details for a range of input data sets, but accurate analysis is difficult if data indirection is used or if the control flow graph is complex. None of these methods adequately provide the capability to predict the memory reference pattern for a broad range of programs and data input sizes.

Characterizing cache behavior has generally taken the form of varying the cache characteristics and measuring behavior for a given program with a particular data input set. The architectural defining features of a cache design are its size, associativity, and cache line (block) size. Techniques have been developed to simulate a wide range of cache sizes [18] and associativities [15] simultaneously. Currently missing is a similar method to efficiently explore a broad range of cache line sizes, though the limited range of line sizes traditionally considered means brute force exploration may be sufficient for the time being.

A fourth dimension little discussed is how cache behavior for a given cache configuration (size, associativity, and line size) varies as the program data set varies. We show this exploration space graphically in Figure 1. The three dimensional space varies cache size on one axis, associativity on another, and program data set size on a third, for a fixed line size. While prior work on cache characterization techniques address how to quickly explore the planar space delineated by the size and associativity axes, this paper describes how to extend the exploration along the program data set size axis in an efficient manner. The method utilizes *data reuse signature patterns*, a fundamental quality of a program behavior [9].

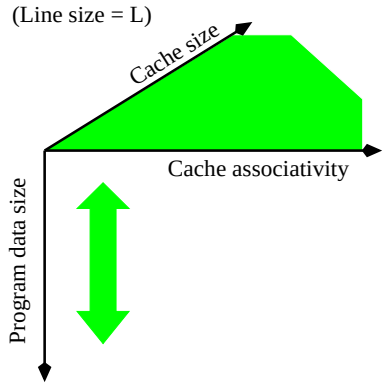


Figure 1. Cache behavior exploration space

Data reuse signature patterns are defined by measurements of a program's *data reuse distance*. In sequential execution, reuse distance is the number of distinct data elements accessed between two consecutive references to the same element. The reuse distance is a measure of the capacity that a fully associative cache must have for the subsequent access to hit in the cache. Thus, reuse distance accurately describes access behavior to a fully associative cache. Ding and Zhong analyzed the reuse pattern of data elements [9]. We use their analysis to analyze the reuse of cache blocks as a pre-step for cache miss prediction.

We present a method for predicting program miss rates across a wide range of program data set and cache sizes. This technique uses reuse distance information from *two* input data sets of different sizes then extracts a parameterized model of the program's fundamental data reuse pattern. We present a method to convert the data reuse distance information into cache miss rates for any input size and for any size of *fully associative cache*. We demonstrate that this method accurately depicts memory reference behavior and cache performance. The predictions are accurate across a wide range of data sets that vary in size by many orders of magnitude. We show the technique provides good approximations even for caches of limited associativity.

A primary strength of this method is that, unlike static compiler analysis, the method is general so it easily accommodates programs with data indirection or complex dynamic control flow. However, not all programs exhibit predictable access patterns, though a surprising number of applications do. Also, the cache blocking factor is not currently part of the model. Thus, all measurements and predictions are made relative to a specified blocking factor, which is 32 bytes in our experiments.

This prediction technique has many uses. First, critical input data set sizes can be identified where the miss rate changes dramatically for a given cache. These *knees* in the miss rate curve have important significance on the overall performance of the program. Second, these critical input data sizes can be well beyond the size of some of the available benchmark reference data sets; thus, the critical data set sizes are unlikely to be discovered by profiling or run-time sampling methods. Third, some applications have inputs which are difficult to generate in various sizes, *e.g.*, *tomcatv* takes a model as input. The predicted miss rate curves provide insight into program behavior that may not be practical to generate by any other means. Fourth, although we focus on sequential applications in this paper, this technique can be applied readily to parallel programming as a guide for partitioning data for efficient caching.

The rest of the paper is organized as follows. In Section 2 we demonstrate how to predict program reference patterns and convert this information to a cache miss rate. Section 3 contains the methodology and results are presented in Section 4. Related work is discussed in Section 5 and we make concluding remarks in Section 6.

2 Cache Miss Rate Estimation

We analyze program locality based on the concept of *reuse distance*. The essence of reuse distance is a measurement of the volume of the intervening data between two accesses. This quality is independent to the granularity of data elements. In this paper, we treat each distinct cache block as a basic data element.

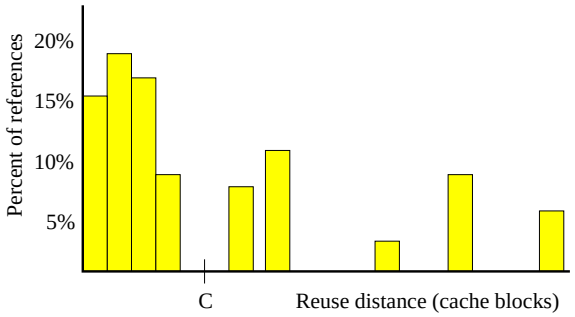


Figure 2. Reuse distance histogram example

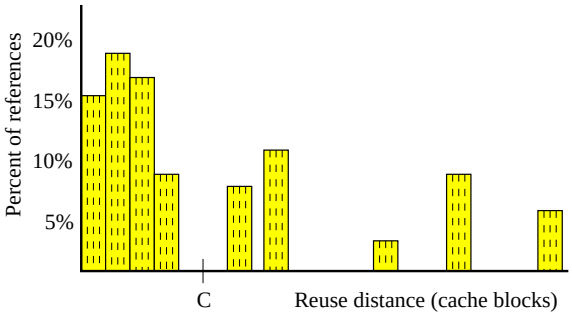
Figure 2 is a histogram of reuse distances for a program. References are grouped by the number of cache blocks referenced between subsequent accesses (*i.e.*, the reuse distance) and the distribution is normalized to the total number of references. The number of intervening cache blocks between two consecutive accesses to the a cache block, along with cache size, determines whether the second access hits in a fully associative LRU cache. This is the basis for our prediction model. In the example, the fraction of references to the left of the mark C will hit in a fully associative cache having capacity of C blocks or greater. For set associative caches, reuse distance is still an important hint to cache behavior[2].

Our approach to predict the cache miss ratio for a given program across different data inputs consists of two main steps. The first step is to model the program locality as predictable reuse distance patterns; the second step is to estimate the miss rate according to the obtained patterns.

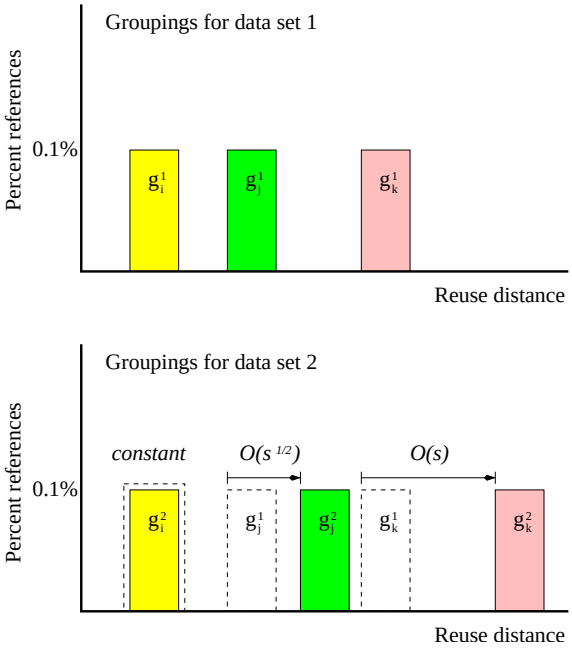
2.1 Locality Pattern Recognition

This section summarizes our prior results from [9] upon which our cache model is based. The reuse histogram depicts locality behavior of a program showing the percentage of memory accesses with a given reuse distance. In profiling, we use Atom[21] to instrument the program and link to the analyzer as described in [9] to collect reuse distance information. Pattern recognition is used to discover the parameterized general model for program locality behavior from the reuse histograms generated during profiling.

Forming reference groups. The reuse distance histogram is composed of references to cache blocks that often exhibit a pattern. The pattern might be a constant reuse distance, *e.g.*, sequential accesses to a block while stepping through an array. The histogram can include patterns that are proportional to the data set size, where the data set size is the number of unique cache lines accessed by the program. For example, reuses of the first access to a block may be related to the square root of the size of a two dimensional array. The histogram is an aggregation of all accesses and,



(a) Partitioning histogram into groups



(b) Pairing groups

Figure 3. Estimating reuse distance functions for each group

thus, includes many types of patterns. The modeling technique classifies portions of the histogram into groups and models each group by a function type.

The difficulty is how to tease apart which references are represented by the different function types. The following technique is a heuristic that works well experimentally. In this technique, we compare the reuse histograms from two runs of the program having two different sizes of data sets. For each histogram, we form $G = 1000$ groups by assigning 0.1% of the histogram to a group, starting from the shortest reuse distance and moving towards the largest. The dotted lines in Figure 3(a) (qualitatively) show a partitioning. An implicit assumption is that references of like reuse distance can be modeled by a similar function type.

To estimate a function for how a group's reuse distance varies with data set size requires at least two sample points for each group. Thus, we pair the i^{th} groups from the two runs, (g_i^1, g_i^2). First, note that the number of references in each run of the program is quite different, but each group represents $\frac{1}{G}$ of the total references of the respective run. Thus, each group has the same *fraction* of references. This presumes that the proportion of each class of reference remains the same for different inputs, which is true for all programs we tested. We justify this pairing of groups with the observation that reuse patterns are a function of data size, s, and this component dominates in determining the reuse distance. The result is that the groups become sorted from left to right by functions of increasing power of s, *e.g.*, reuse distance increases more rapidly for groups based on s than for those based on $\sqrt{s}$.

In Figure 3(b) we show three groups and the reuse distance histogram for two runs. In the second run, the movement of the groups is shown relative to the first run. The estimated function types are also shown. The first group g_i does not move between the runs, so its reuse distance is independent of the data set size (*constant*). On the other hand, g_j has a square root relationship to the data size, $O(\sqrt{s})$, and g_k has a linear relationship, $O(s)$. Dividing the histogram into 1000 groups limits the error from any single estimated model. The final model is a parameterized set of 1000 functions. The method of determining the functions is presented next.

Recognizing individual patterns. Given two histograms with different data sizes, the pattern recognition step constructs a formula for each group of references. Let d_i^1 be the distance of the i^{th} group in the first histogram, d_i^2 be the distance of the i^{th} group in the second histogram, s_1 be the data size of the first run, and s_2 the data size of the second run. The pattern for this group is a linear fitting based on the data size. Specifically, we want to find the two coefficients, c_i and e_i, that satisfy the following two equations.

$$d_i^1 = c_i + e_i * f_i(s_1) \quad (1)$$

$$d_i^2 = c_i + e_i * f_i(s_2) \quad (2)$$

Assuming the function f_i is known, the two coefficients uniquely determine the distance for any other data size. The pattern is more accurate if more profiles are collected for the linear fitting. The minimal number of inputs is two.

The formula is based on an important fact about reuse distance: in any program, *the largest reuse distance cannot exceed the size of program data*. Therefore, the function f_i can be linear at most, and the pattern is a linear or sub-linear function of data size, not a general polynomial function. We consider the following choices for f_i. The first is $p_{const}(s) = 0$. We call such formula a constant pattern because reuse distance does not change with data size. The second is $p_{linear}(s) = s$, a linear pattern. Constant and linear are the lower and upper bounds of the distance patterns. Between them are sub-linear patterns, for which we consider three: $p_{1/2}(s) = s^{1/2}$, $p_{1/3}(s) = s^{1/3}$, and $p_{2/3}(s) = s^{2/3}$. The first occurs in two dimensional problems such as matrix computation. The other two occur in three dimensional problems such as physics simulation. We

could consider higher dimension problems in the same way but we did not find a need in the programs we tested.

For each group pair (g_i^1, g_i^2), we calculate the ratio of their average distance, d_i^2/d_i^1, and pick f_i to be the pattern function, p_t, such that $p_t(s_2)/p_t(s_1)$ is closest to d_i^2/d_i^1. Here t is one of the patterns described in the last paragraph.

Recording the overall model. We select large enough sizes of profiling inputs so that it is unlikely that references having different patterns (i.e., functions based on input size s) become mixed in the same group. The overall model of the program is the aggregation of the individual fitted functions for each group. The details for the set of functions making up the model are recorded in the data structures defined in Figure 4.

```
model for program p
    Model = structure{
              patternNo: number of patterns;
              patterns: Pattern[patternNo];
              groupNo: total number of groups;
            }
    Pattern = structure{
              type: enum{ p_const, p_1/3,
                          p_1/2, p_2/3, p_linear }
              groupNo: number of groups;
              groups: Formula[groupNo];
            }
    Formula = structure{
              c: constant coefficient;
              e: non-constant coefficient;
            }
```

Figure 4. Locality Model Definition

2.2 Cache Miss Rate Estimation

After generating the locality model of a program, we can use the model to predict the locality behavior for any data input size s_d. We predict the reuse distance for each group g_i via their particular formula given s_d:

$$d_i = c_i + e_i * f_i(s_d) \quad (3)$$

The overall reuse distance distribution is the aggregation of the reuse distance of each group. For any given size of a fully associative LRU cache, we estimate the capacity miss rate directly from the reuse distance distribution. The groups with a reuse distance larger than or equal to the given cache size will be capacity misses. The number of these groups gives an estimation of the miss rate. The detailed algorithms for reuse distance prediction and cache miss rate estimation are described in Figure 5.

From the locality model, we can also derive the maximum possible miss rate of a program for a given cache size. Here we consider two important properties of the reuse distance model. First, only references with a reuse distance larger than or equal to the cache size are cache misses. References with shorter reuse distance will hit in cache because of temporal locality. The second property

```
algorithm PredictRD(model, size_d)
  input: model: the locality model
         size_d: the data input size
  output: rd[model.groupNo]:
          the reuse distance distribution
  begin
     index = 0;
     foreach pattern in model.patterns do
       f = pattern.type;
       foreach formula in pattern.groups do
          distance = formula.c + f(size_d) * formula.e;
          rd[index]=distance;   index++;
       end foreach
     end foreach
  end
end algorithm

algorithm EstimateMR(model, size_d, size_c)
  input: model: the locality model
         size_d: the data input size
         size_c: the cache size
  output: missRate
  begin
     missed =0;
     rd = PredictRD(model,size_d);
     for (index = 0 ; index<model.groupNo;index++)
       if (rd[index] ≥ size_c)
          missed ++;
     end for
     missRate = missed / model.groupNo;
  end
end algorithm
```

Figure 5. Cache miss rate estimation

is that reuse distance having linear or sub-linear patterns monotonically increase with the input size. The distances of references having a constant pattern, on the other hand, are independent of the input size. From these two facts, we can infer that for a certain cache size, if we increase the program data input size, the cache miss rate may remain the same or increase, but it will never decrease. Furthermore, the maximum miss rate will be reached when the input size is large enough so that all references with a non-constant pattern have a reuse distance larger than the cache size.

Figure 6 shows the main algorithm to predict the maximum cache miss rate and the corresponding threshold input size for a given cache of size C blocks. Suppose among the total G groups ($G = 1000$ for all our results), the number of reference groups with a pattern dependent on data size s is G_s and the number of groups having a pattern independent of data size (constant) with a fixed reuse distance greater than the capacity of the cache is G_c, the maximum miss rate is calculated as $Miss\ Rate_{max} = (G_s + G_c)/G$.

The *threshold input size (T_s)* is the smallest input size in which cache miss rate reaches the maximum value for the program and given cache configuration. To estimate its value, we only need to consider the single non-constant reference group having the shortest reuse distance. The maximum miss rate occurs when this group, g_j, has a reuse distance greater than or equal to the cache size C, $d_j \geq C$.

```
algorithm MaxMR(model, size_c)
 input: model: the locality model
        size_c: the cache size
 output: maxMR: maximum miss rate
         size_d: threshold data input size
 begin
    constPattern = model.patterns[0];
    nonConstGroup =
        model.groupNo - constPattern.groupNo;
    constGroup = ConstMR(model,size_c);
    maxMR = (constGroup + nonConstGroup) /
                model.groupNo;
    if (model.patternNo > 1)
      size_d =
        GetCriticalInputSize(model.patterns[1],size_c);
    else
      size_d = null;
    end if
 end
end algorithm

subroutine ConstMR(model, size_c)
 input: model: the locality model
        size_c: the cache size
 output: constGroup: the number of constant groups
         with a reuse distance longer than size_c
 begin
    index =0;
    constPattern = model.patterns[0];
    constRD = constPattern.groups[0].c;
    while (size_c ≤ constRD) do
      index++;
      constRD = constPattern.groups[index].c;
    end while
    constGroup = constPattern.groupNo - index - 1;
 end
end subroutine ConstMR

subroutine GetCriticalInputSize(pattern, distance)
 input: pattern: the given pattern of model
        distance: the reuse distance
 output: size_d: threshold data input size
 begin
    f = pattern.type;
    formula = pattern.groups[0];
    size_d =
        f^{-1}((distance − formula.c)/formula.e);
 end
end subroutine GetCriticalInputSize
```

Figure 6. Max miss rate & threshold input prediction

By manipulating Equation 3 we can directly calculate the required input data size from this condition:

$$T_s = f_j^{-1}((C - c_j)/e_j) \quad (4)$$

If the model only contains the constant pattern, the cache miss rate is the same for all inputs. In this case, e_j is zero and generates a meaningless null threshold input. The threshold input size T_s is useful since it is the smallest input that generates the worst case hit ratio for a fully associative cache (we address limited associativity in Section 4).

Model accuracy verification. Only two input sets are necessary to generate a miss rate model for a program. However, if more input sets are measured, multiple models can be generated from pair-wise groupings and the predictions evaluated for the excluded data set sizes. For example, for three data set sizes, A, B, and C, three combinations of pairs are possible, *(A, B)*, *(A, C)*, and *(B, C)*. The data sets used for verification would be C, B, and A, respectively. Thus, a degree of confidence for the model can be measured with limited exploration of the data set space.

3 Methodology

We use the cache simulator *Cheetah*[22] included in Simplescalar 3.0 toolset to collect cache miss statistics. Cache configurations are fully associative cache, 1-, 2-, 4- and 8-ways and all with a 32-byte block size. We use *Atom*[21] to instrument the binary to collect the addresses of all loads and stores and feed them to Cheetah.

The model predicts capacity miss ratios. The fraction of compulsory misses is the ratio of the data touched to the total number of accesses. This ratio cannot be predicted, in general, because of data dependent features such as a variable number of iterations over the data. However, the relative fraction of compulsory misses is small when there is significant data reuse and can be ignored without significant impact on the miss rate. In our results, the compulsory misses make up an average of 0.7% of the accesses and account for less than 10% of the misses.

To evaluate our prediction, we post-process the data collected by Cheetah and extract the number of capacity and conflict misses, but exclude the compulsory misses. We call the extracted miss rate the *reuse miss rate* and calculate it from the number of accesses (N_{total}) and the number of compulsory misses ($N_{compulsory}$):

$$reuse\ miss\ rate = \frac{miss\ rate \times N_{total} - N_{compulsory}}{N_{total} - N_{compulsory}}$$

In the results, all reported miss rates are the *reuse miss rate*. The sizes of the two target caches are 64 KB and 1 MB, which represent reasonable sizes for the first and second level caches, respectively.

The benchmarks are listed in Table 1. The third column shows the main patterns identified for each of the programs. The table also shows the accuracy of our prediction scheme for reuse distance. Let x_i and y_i be the size of i^{th} group in predicted and measured histograms. The cumulative difference, E, is the sum of differences $|y_i - x_i|$ for all i. In the worst case, E is 200%. We use $1 - E/2$ as the accuracy. It measures the common parts of the two histograms, ranging from 0% (no match) to 100% (perfect match). To show the accuracy of our prediction of reuse distance, we choose three inputs for each benchmark, as listed in column 4. Column 5 and 6 summarize these inputs by data input size in cache blocks and the total number

Table 1. Cache Block Reuse Pattern Prediction

Benchmark	Description	Patterns	Input	Data size in cache blocks	Memory references	Block Accuracy(%)	Element * Accuracy(%)
Applu	solution of five	const	ref(60^3)	5.70M	4.23B	**96.1**	**83.6**
(Spec2K)	coupled nonlinear	3rd roots	train(24^3)	333K	230M	**97.0**	**94.7**
	PDE's	linear	test(12^3)	34.5K	22.3M		
Swim	finite difference	const	ref(512^2)	461K	132M	**98.9**	**99.0**
(Spec95)	approximations for	2nd root	400^2	289K	80.7M	**98.7**	**99.3**
	shallow water equation	linear	200^2	74.3K	20.2M		
SP	computational fluid	const	50^3	1.21M	443M	**96.5**	**90.3**
(NAS)	dynamics (CFD)	3rd roots	32^3	323K	110M	**97.0**	**95.8**
	simulation	linear	28^3	218K	74.4M		
Tomcatv	vectorized mesh	const	ref(513^2)	459K	39.4M	**94.3**	**92.4**
(Spec95)	generation	2nd root	train(257^2)	116K	9.83M		
		linear	400^2	282K	39.1M	**94.6**	**99.2**
FFT	fast Fourier	const	512^2	262K	67.8M	**91.8**	**84.3**
	transformation	2nd root	256^2	65.6K	15.3M	**94.0**	**94.0**
		linear	128^2	16.5K	3.45M		
Gcc	based on the		cp-decl	86.6K	134M	**98.7**	**98.6**
(Spec95)	GNU C compiler	const	amptjp	53.4K	103M	**98.7**	**98.7**
	version 2.5.3		genoutput	18.2K	9.71M		
ADI	two-dimensional	const	500^2	188K	5.99M	**97.5**	**na**
	alternate direction	2nd root	300^2	67.5K	2.16M	**94.3**	**na**
	implicit integration	linear	200^2	30.0K	959K		
					Average	**96.3**	**94.2**

*Element accuracy comes from [9](PLDI'03)

of memory references. Based on the profiled results of the two smaller inputs, we predict the reuse distance distribution for the largest input. We also predict the middle one based on the smallest and largest inputs. The results are given in the second to the last column. The average accuracy is 96.3%. In our earlier paper[9], we report the prediction of reuse distance for each data element and reprint those results in the last column. The last two columns show the block-based prediction obtains comparable results.

4 Results

4.1 Fully-associative cache results

Shown in Figures 7 and 8 are the predicted cache miss curves for each of the seven applications assuming a fully associative cache. In each pairing, the upper graph is the estimate for a 64KB (2K blocks) cache and the lower graph is the estimate for a 1MB (32K blocks) cache. The input data set size is varied along the x-axis and the miss rate percentage along the y-axis. The data size is given in *cache blocks*; multiplying by 32 converts the range to bytes. The data set size is the number of unique cache lines accessed by the program. The range on the y-axis is the same in all graphs, but the range on the x-axis varies and is shown as $\log$ scale. The two leftmost vertical lines shown in each graph mark the two sample data set sizes to generate the model. The third line is the largest input data size we were able to simulate. The vertical lines are consistent (at the same data sizes) within a pair of graphs.

The most noticeable feature is that each graph has one or more inputs in which the miss rate exhibits a sharp jump. These jumps occur at input data set sizes where another portion of the reuse histogram dependent on s moves beyond the cache size. Thus, the maximum miss rate of an application occurs further out in the 1 MB cache compared to the 64KB cache. At input data sizes that achieve the maximum miss rate, all hits in the cache are due to the portion of the reuse distance histogram that have a constant reuse distance within the cache size. It is interesting to note that the estimation function does not require the input data sets be from sizes corresponding to "interesting" regions of the miss rate graph, *e.g.*, bracketing jumps in the miss rate.

The application *gcc* is notable in its lack of features. The constant pattern has the best fit because the working data set size is related to the per function size. Since the input programs all have similarly sized functions (though programs may have a different number of functions), the reuse distance pattern is insensitive to the total program size. The graphs show a horizontal line for both the 64 KB and 1 MB caches (the line is on the zero mark).

4.2 Limited associativity cache results

Reuse distance is based on a fully associative cache design implementing LRU replacement. However, fully associative hardware caches are impractical to implement, so the caches typically have limited associativity from direct-mapped to 8-way. In Figures 9 and 10, we verify the cache miss rate predictions of our model to actual miss rates from detailed cache simulations. We show the results for each application for both 64 KB and 1 MB cache sizes. Each graph has six curves plotted: direct-mapped, 2-way, 4-way, 8-way, fully associative, and the predicted miss rate.

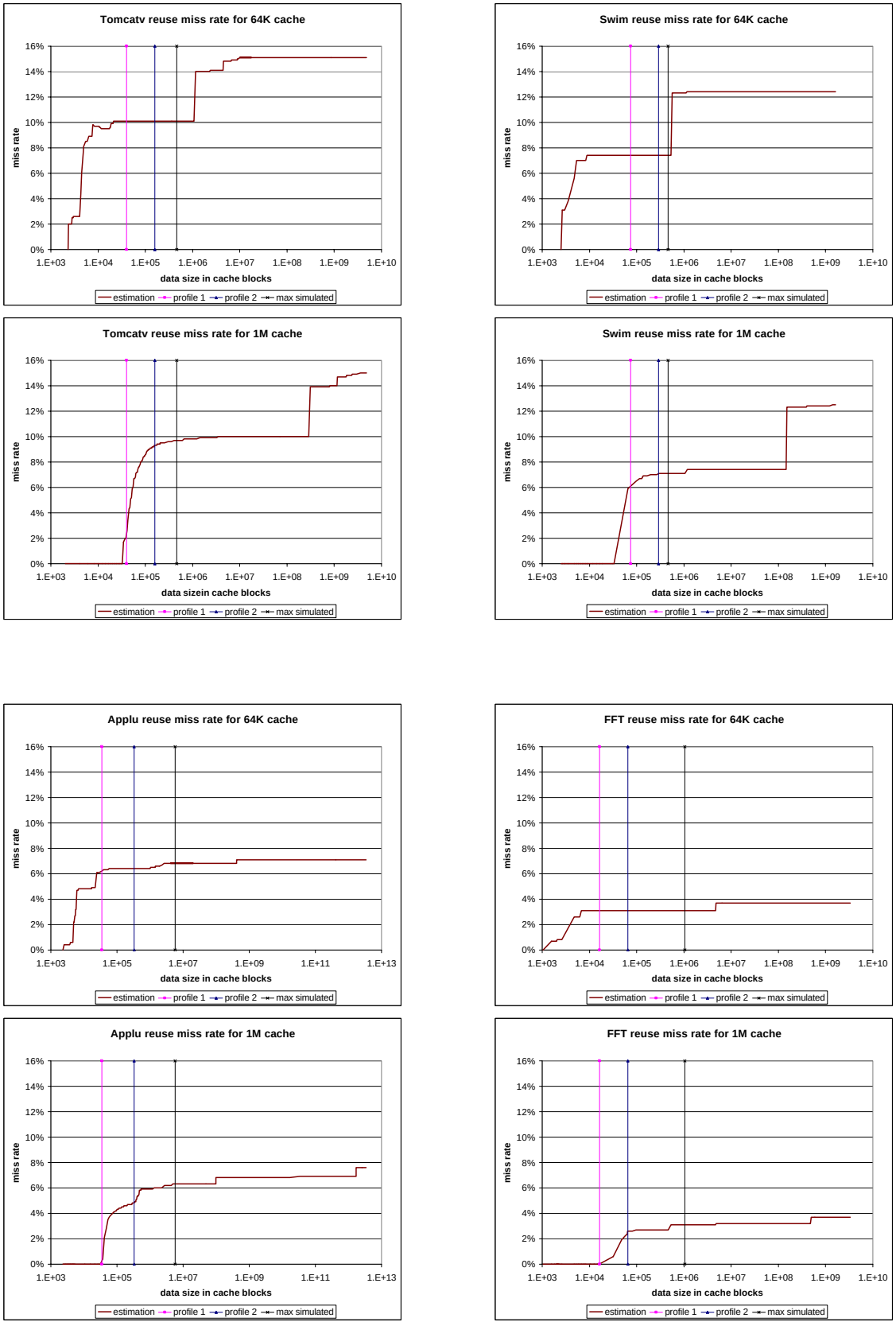


Figure 7. Estimated miss rates for *tomactv, swim, applu, fft*, 64KB and 1MB caches

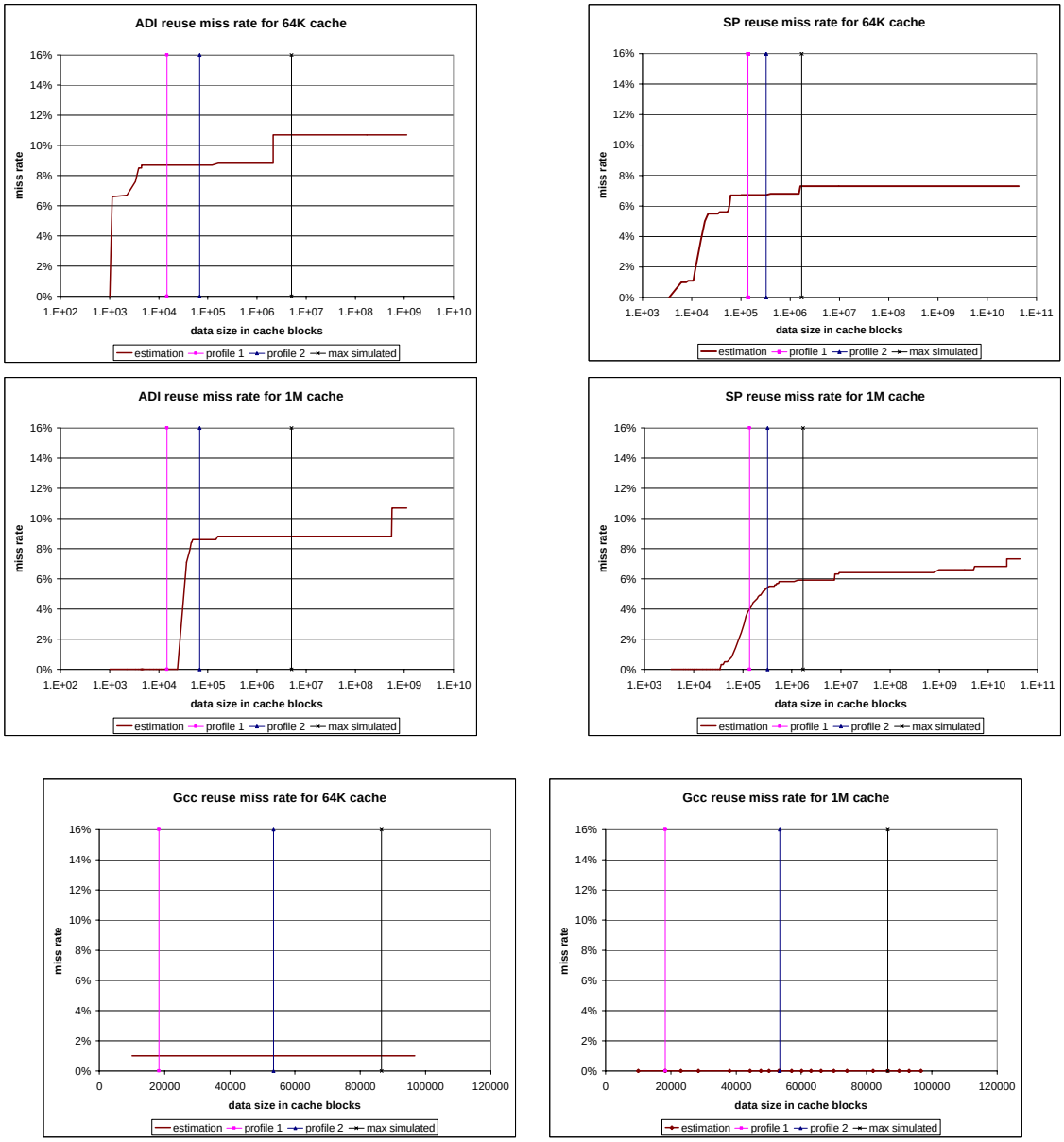


Figure 8. Estimated miss rates for *ADI, SP, gcc*, 64KB and 1MB caches

The graphs in Figures 9 and 10 use a subset of the ranges for the predictions shown in Figures 7 and 8. The smaller range is because of limitations in the processor memory, the capabilities of the Cheetah simulator, or the difficulty in generating large enough input sets. In Figures 7 and 8, the third vertical line (rightmost) in each graph marks the largest data set size actually simulated. What is evident is that most applications exhibit interesting behavior beyond what we were able to simulate for this paper. This difficulty highlights a benefit of our approach: a parameterized model of cache behavior can reveal data set sizes where interesting cache behavior occurs and where program analysis should be focused. In fact, only for *ADI* with a 64 KB cache were we successful in simulating a wide enough data set range to capture all interesting cache behavior predicted by the model.

Overall, it is difficult to separate the different lines in each of the plots because the curves across the different configurations are so similar. However, this also demonstrates that our modeling technique makes accurate predictions. Figures 11(a) and 11(b) summarize the relative error across all the applications and cache configurations. Shown is the absolute relative error of the *hit rate* between the measured and predicted values averaged across all measured data points for each application and for each associativity. We use the hit rate for the error measure because it is more

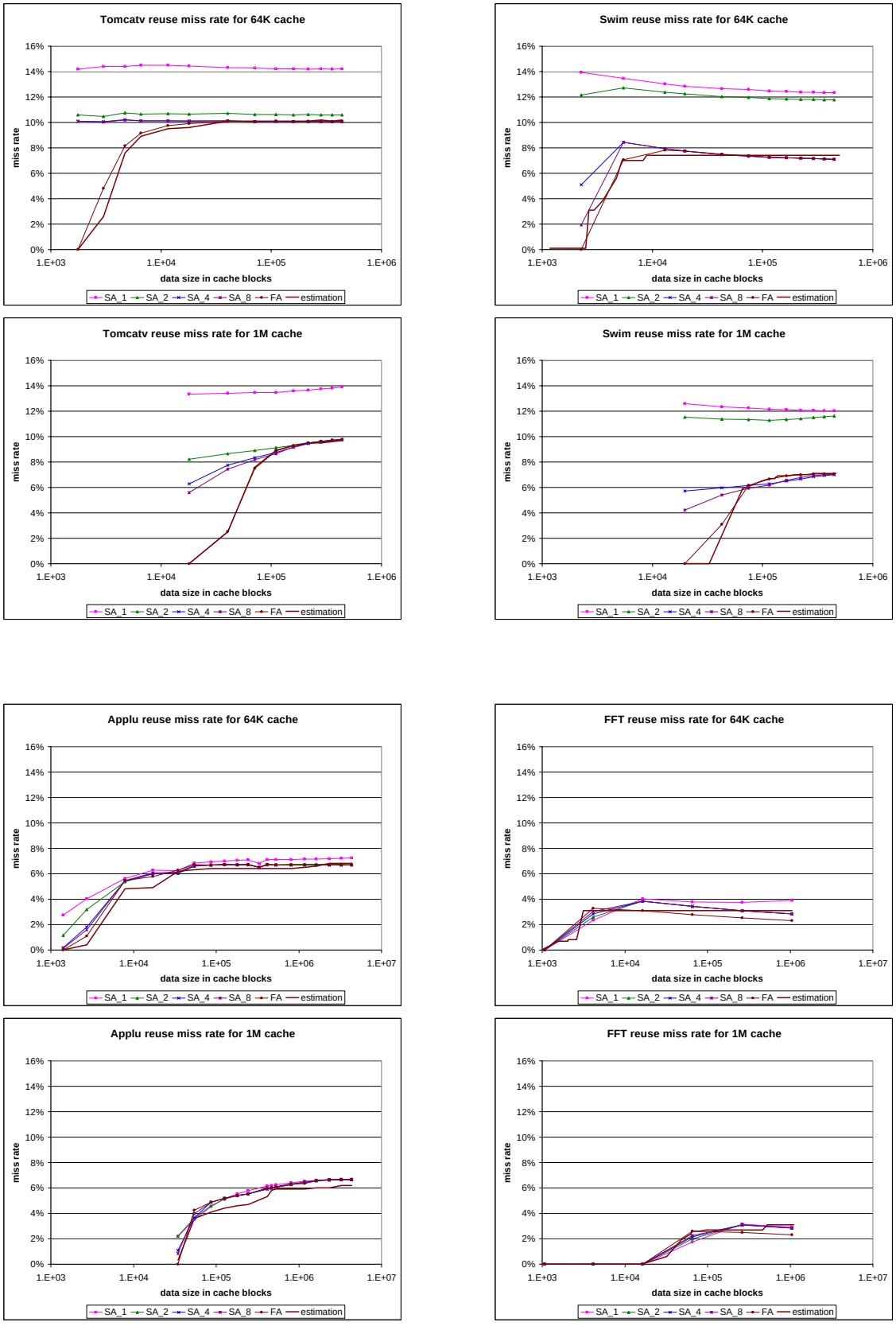


Figure 9. Verification miss rate estimates *tomactv, swim, applu, fft*, 64KB and 1MB caches

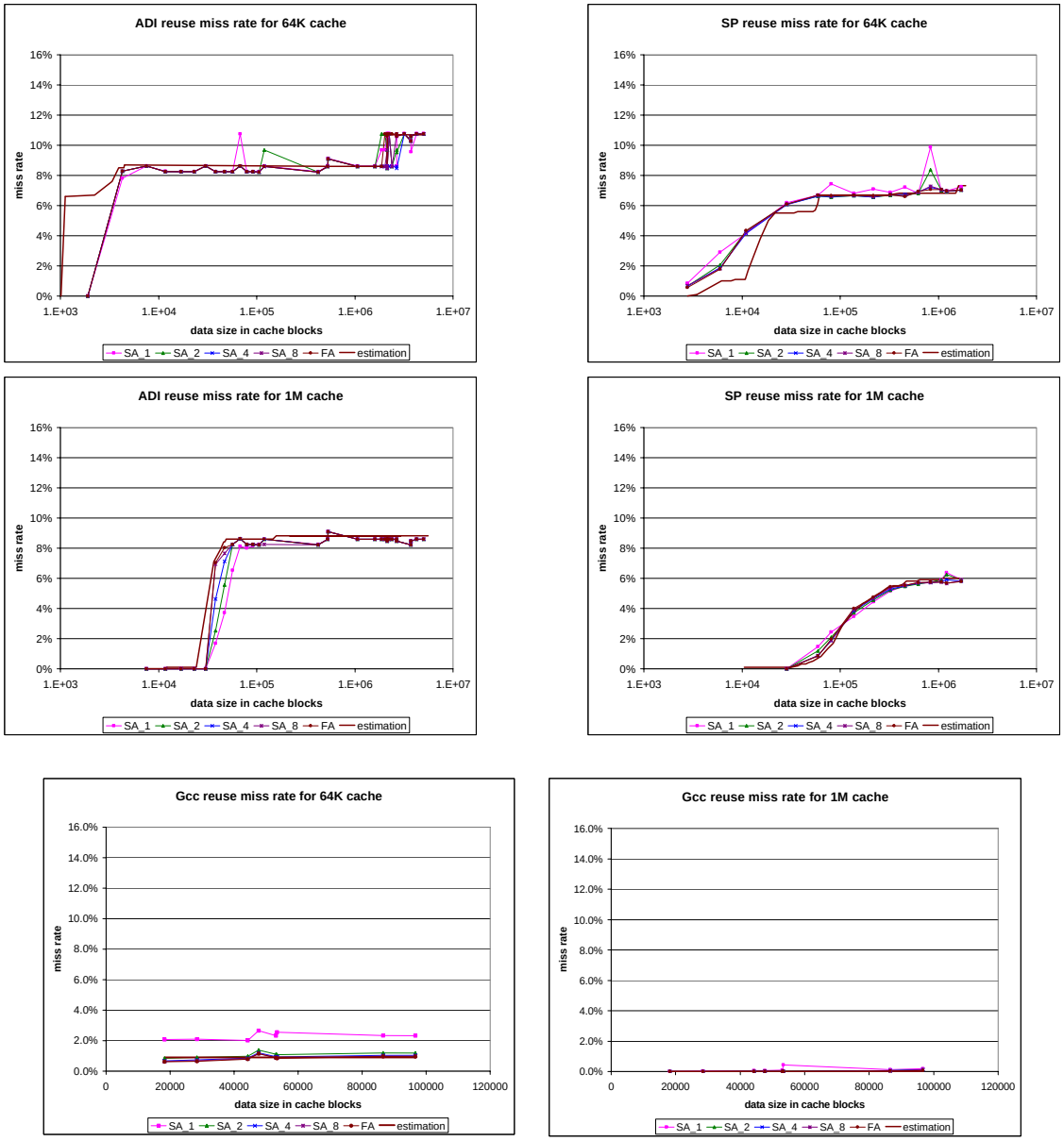


Figure 10. Verification miss rate estimates *ADI, SP, gcc*, 64KB and 1MB caches

stable than the miss rate in regions where the miss rate is near zero. Of course, a small percentage error in the hit rate will generally represent a larger relative percentage in the miss rate. The full data set shown in Figures 9 and 10 can be used to put the error rates into proper perspective.

From Figure 11, the average relative error is less than 1% between the model and a simulated fully associative cache (FA). The error is below 2% for all applications when the cache associativity is 4-way or greater. Two applications, *tomcatv* and *swim*, have a higher relative error (max 7.4%) when the associativity is small. Referring back to Figures 9 and 10, the direct mapped cache plot for both applications have noticeably worse miss rates. These additional misses are conflict misses which full associativity eliminates. The overall accuracy despite ignoring conflict misses is due to the fact in many applications that the miss rate is dominated by capacity misses [19].

The program *ADI* exhibits interesting behavior in Figure 10. For a cache size of 1 MB, the model predicts a dramatic rise in the miss rate between data set sizes of 20K and 40K cache blocks, *i.e.*, 640 KB and 1.28 MB. This is the point in which the cache's 1 MB capacity is exceeded. However, the direct mapped cache's miss rate exhibits a slower rise between 30K and 70K cache blocks, *i.e.*, 960 KB and 2.1 MB. The reason for the improved miss rate is that the fully associative cache LRU policy displaces blocks

that will be used in the relatively near future because of insufficient capacity. In the direct mapped cache, however, the limited mapping of blocks results in blocks being replaced randomly relative to their access order, which happens to improve the miss rate overall for a limited data set range.

A visualization tool Interested readers can recreate these results as well as results for other cache sizes and other benchmarks using an on-line interactive graphical tool at `www.cs.rochester.edu/research/locality/`.

5 Related Work

Program cache behavior has been extensively studied mainly from two directions: program analysis and trace-driven cache simulation.

Program analysis. Dependence analysis analyzes data accesses and can measure the number of capacity misses in a loop nest. Gannon *et al.* estimated the amount of memory needed by a loop [12]. Other researchers used various types of array sections to measure data access in loops and procedures. Such analysis includes linearization for high-dimensional arrays by Burke and Cytron [4], linear inequalities for convex sections by Triolet *et al.* [23], regular sections by Callahan and Kennedy [5], and reference list by Li *et al.* [17]. Havlak and Kennedy studied the effect of array section analysis on a wide range of programs [14]. Cascaval extended dependence analysis to estimate the distance of data reuses [6]. One limitation of dependence analysis is that it does not model cache interference caused by the data layout. Ferrente *et al.* gave an efficient approximation of cache interference in a loop nest [11]. Recent studies used more expensive (worst-case super-exponential) tools to find the exact number of cache misses. Ghosh *et al.* modeled cache misses of a perfect loop nest as solutions to a set of linear Diophantine equations [13]. Chatterjee *et al.* studied solutions of general integer equations and used Presburger solvers like Omega [7]. The precise methods are effective for a single loop nest. For full applications, researchers have combined compiler analysis with cache simulation. McKinley and Temam carefully measured various types of reference locality within and between loop nests [19]. Mellor-Crummey *et al.* measured fine-grained reuse and program balance through a tool called HPCView [20].

For regular loop nests, compiler analysis identifies not only the cache behavior but also its exact causes in the code. However, the precise analysis does not yet model inter-nest cache misses efficiently except for Cascaval [6]. In addition, compiler analysis is not as effective for programs with input-dependent control flows and data indirection. This paper presents an alternative that is fairly accurate, efficient, and applicable to programs with arbitrary control flow and data access expressions. At least four compiler groups have used reuse distance for different purposes: to study the limit of register reuse [16] and cache reuse [8, 24], to evaluate the effect of program transformations [1, 2, 8, 24], and to annotate programs with cache hints to a processor [3]. In the last work, Beyls and D'Hollander used reuse distance profiles to generate hints in SPEC95 FP benchmarks and improved performance by 7% on an Itanium processor [3]. The techniques in this paper will allow compiler writers to estimate cache behavior for data inputs based on a few profiling runs.

Cache simulation. Trace-driven cache simulation has been the primary tool to evaluate cache design including the cache size, block size, associativity, and replacement policy. Mattson *et al.* gave a stack algorithm that measured cache misses for all cache sizes in one simulation [18]. Hill extended the one-pass algorithms to measure the miss rate in direct-mapped caches and practical set-associative caches [15]. While these techniques simulated the entire address trace, many later studies used sampling to reduce the length of the simulation. Our work adds another dimension. It estimates the miss rate of data inputs without running *all* inputs, including those that might be too large to run, let alone to simulate.

Eeckhout *et al.* studied correlations between miss rates of 9 programs across 79 inputs using principal components analysis followed by hierarchical clustering [10]. Since program runs in the same cluster had similar behavior, they could reduce the redundancy in a benchmark set by picking only one run from each cluster. Our result may strengthen their method by increasing the coverage. It suggests that many programs have only a few different miss rates across all data inputs, and a wide range of inputs may have the same miss rate. The miss-rate prediction can ensure that a benchmark set includes all miss rates and the smallest program runs for these miss rates by suggesting them as candidates for benchmark selection.

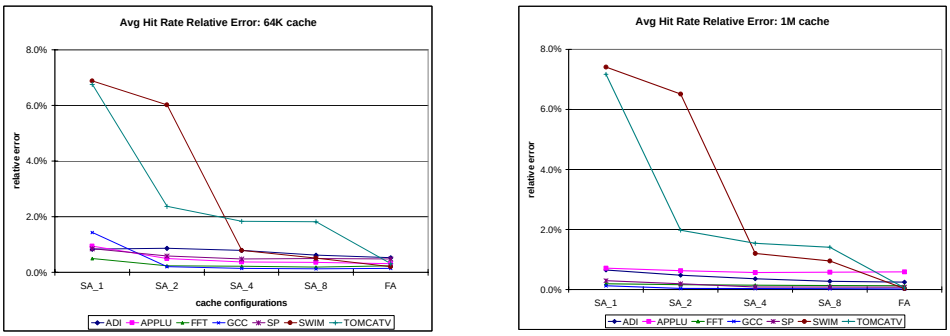
6 Summary

This paper presents an algorithm for estimating the miss rate of a program for all its input sizes. Based on a few training runs, the algorithm constructs a parameterized model that predicts the miss rate of a fully associative cache with a given cache-block size. By supplying the range of all input sizes as the parameter, we can predict miss rates for all data inputs on all sizes of fully associative caches. For a given cache size, the model also predicts the input size where the miss rate exhibits marked changes. Our experiments show the prediction accuracy is 99% for fully associative caches and better than 98% for caches of limited associativity, excluding compulsory misses. In addition, the predicted miss rate is either very close or proportional to the miss rate of direct-map or set-associative cache.

References

[1] G. Almasi, C. Cascaval, and D. Padua. Calculating stack distances efficiently. In *Proceedings of the first ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, June 2002.

[2] K. Beyls and E. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, August 2001.



(a) 64KB cache

(b) 1MB cache

Figure 11. Relative hit rate error

[3] K. Beyls and E. D'Hollander. Reuse distance-based cache hint selection. In *Proceedings of the 8th International Euro-Par Conference*, Paderborn, Germany, August 2002.

[4] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[5] D. Callahan, J. Cocke, and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5(5):517–550, Oct. 1988.

[6] G. C. Cascaval. *Compile-time Performance Prediction of Scientific Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 2000.

[7] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, UT, 2001.

[8] C. Ding. *Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse*. PhD thesis, Dept. of Computer Science, Rice University, January 2000.

[9] C. Ding and Y. Zhong. Predicting whole-program locality with reuse distance analysis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

[10] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Workload design: selecting representative program-input pairs. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, Virginia, September 2002.

[11] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, Aug. 1991. Springer-Verlag.

[12] K. Gallivan, W. Jalby, and D. Gannon. On the problem of optimizing data transfers for complex memory systems. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[13] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Langauges and Systems*, 21(4), 1999.

[14] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[15] M. D. Hill. *Aspects of cache memory and instruction buffer performance*. PhD thesis, University of California, Berkeley, November 1987.

[16] Z. Li, J. Gu, and G. Lee. An evaluation of the potential benefits of register allocation for array references. In *Workshop on Interaction between Compilers and Computer Architectures in conjuction with the HPCA-2*, San Jose, California, February 1996.

[17] Z. Li, P. Yew, and C. Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):26–34, Jan. 1990.

[18] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[19] K. S. McKinley and O. Temam. Quantifying loop nest locality using SPEC'95 and the perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4):288–336, November 1999.

[20] J. Mellor-Crummey, R. Fowler, and D. B. Whalley. Tools for application-oriented performance tuning. In *Proceedings of the 15th ACM International Conference on Supercomputing*, Sorrento, Italy, June 2001.

[21] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, Florida, June 1994.

[22] R. A. Sugumar and S. G. Abraham. Efficient simulation of multiple cache configurations using binomial trees. Technical Report CSE-TR-111-91, CSE Division, University of Michigan, 1991.

[23] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of CALL statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[24] Y. Zhong, C. Ding, and K. Kennedy. Reuse distance analysis for scientific programs. In *Proceedings of Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Washington DC, March 2002.