Miss Rate Prediction Across Program Inputs and Cache Configurations

Yutao Zhong, *Member*, *IEEE Computer Society*, Steven G. Dropsho, *Member*, *IEEE*, Xipeng Shen, Ahren Studer, and Chen Ding, *Member*, *IEEE Computer Society*

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 and all cache configurations. This paper uses 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 percent of the hit rate for set associative caches on a set of floating-point and integer programs using array and pointer-based data structures. Building on the new model, this paper presents an interactive visualization tool that uses a three-dimensional plot to show miss rate changes across program data sizes and cache sizes and its use in evaluating compiler transformations. Other uses of this visualization tool include assisting machine and benchmark-set design. The tool can be accessed on the Web at http://www.cs.rochester.edu/research/locality.

Index Terms—Cache memories, modeling techniques, performance analysis and design aids, compilers, optimization.

1 Introduction

The efficiency of a computer system in executing a program is highly dependent 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. Past work mainly provides three ways of locality analysis: by a compiler, which models loop nests but is not as effective for dynamic control flow and data indirection; by frequency profiling, which analyzes a program for select inputs but does not predict the behavior change in other inputs; or by runtime analysis, which cannot afford to analyze every access to every data. None of these methods adequately provides the capability of predicting the memory reference patterns across 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 architecturally 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 [1] and associativities [2] 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 Fig. 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 addresses 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 program behavior [3].

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. We analyze the reuse pattern of cache blocks as a prestep for cache miss prediction.

- *Y. Zhong is with the Department of Computer Science, George Mason University, Fairfax, VA 22030. E-mail: yzhong@cs.gmu.edu.*
- *S.G. Dropsho is with the Swiss Institute of Technology (EPFL), Lausanne, Switzerland. E-mail: steven.dropsho@epfl.ch.*
- *X. Shen is with the Department of Computer Science, College of William and Mary, Williamsburg, VA 23185. E-mail: xshen@cs.wm.edu.*
- *A. Studer is with Carnegie Mellon University, Pittsburgh, PA 15213. E-mail: astuder@andrew.cmu.edu.*
- *C. Ding is with the Department of Computer Science, University of Rochester, Rochester, NY 14627. E-mail: cding@cs.rochester.edu.*

Manuscript received 23 Dec. 2003; revised 11 May 2005; accepted 1 June 2005; published online 22 Jan. 2007.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0294-1203.

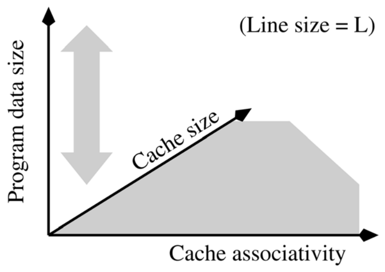


Fig. 1. Cache behavior exploration space.

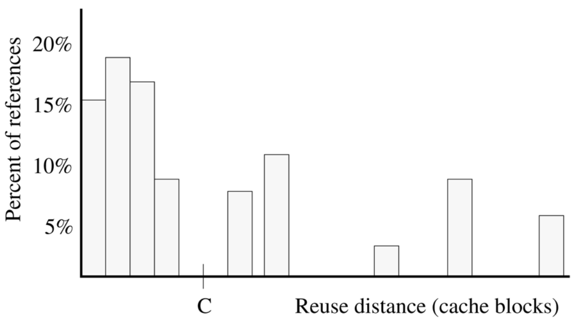


Fig. 2. Reuse distance histogram example.

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. Regression with additional training input data sets can further improve the accuracy of the model. 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.

Building on the new model, this paper presents a visualization tool that uses three-dimensional plots to show how the miss rate changes across program input sizes and cache sizes. The tool has many uses. First, critical input data set sizes can be identified where the miss rate changes dramatically for a given cache. 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 runtime 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 in a parallel environment to model the memory behavior of individual processes or threads.

This paper demonstrates the visualization tool in compiler evaluation. It measures the effect of a program transformation not just for one program execution on a single machine but for all program inputs for all cache configurations. In addition, the paper discusses other uses of the tool in cost-effective cache design and benchmark set design.

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 measures the accuracy of miss rate prediction. Section 4 presents the visualization tool and its uses. Related work is discussed in Section 5 and we conclude 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, which can be approximated in near constant time for each access with a guaranteed accuracy [3]. This quality is independent of the granularity of data elements. In this paper, we treat each distinct cache block as a basic data element.

Fig. 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 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 Fig. 2, 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 [4].

Our approach to predicting 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 [3], [5], upon which our cache model is based. The reuse histogram depicts the locality behavior of a program showing the percentage of memory accesses with a given reuse distance. In profiling, we use Atom [6] to instrument the program and link to the analyzer, as described in [3], 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.

2.1.1 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, 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 $G = 1,000$ groups by assigning 0.1 percent of the histogram to a group, starting from the shortest reuse distance and moving toward the largest. The dotted lines in Fig. 3a (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 ith 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 largely true for all programs we tested, as indicated by the experimental data presented in Section 3. 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. For a nontrivial data size, 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 Fig. 3b, 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 1,000 groups limits the error from any single estimated model. The final model is a parameterized set of 1,000 functions. The method of determining the functions is presented next.

2.1.2 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 ith group in the first histogram, d_i^2 be the distance of the ith 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:

$$d_i^1 = c_i + e_i * f_i(s_1), \tag{1}$$

$$d_i^2 = c_i + e_i * f_i(s_2). \tag{2}$$

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, so the pattern is a linear or sublinear function of data size and not a general polynomial function. We consider the following choices for f_i: The first is $p_{const}(s) = 0$. We call such a formula a constant pattern because reuse distance does not change with data size. The

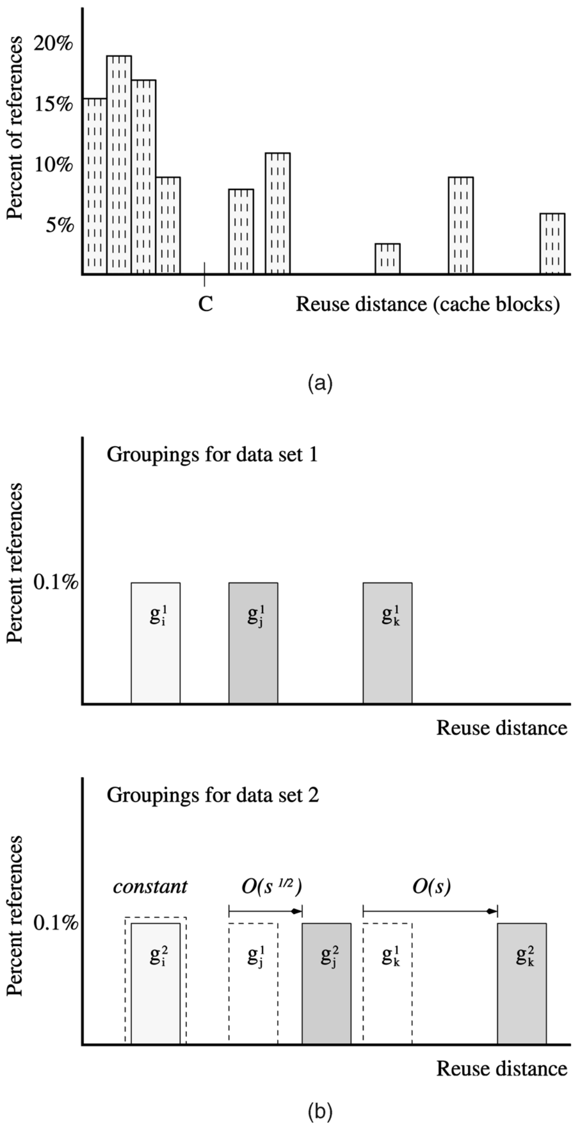


Fig. 3. Estimating reuse distance functions for each group. (a) Partitionings histogram into groups. (b) Pairing groups.

```
Model = structure{
            groupNo: total number of groups;
            formulas: Formula[groupNo];
        }
Formula = structure{
            c: constant coefficient;
            e: non-constant coefficient;
            pattern: Pattern;
        }
Pattern = enum{ $p_{const}$, $p_{1/3}$, $p_{1/2}$, $p_{2/3}$, $p_{linear}$ }
```

Fig. 4. Locality model definition.

second is $p_{linear}(s) = s$, a linear pattern. The constant and linear patterns are the lower and upper bounds of the distance patterns. Between them are sublinear 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, such that $p(s_2)/p(s_1)$ is closest to d_i^2/d_i^1. Here, p is one of the patterns described in the last paragraph.

2.1.3 Resolving Coefficients Based on Regression

According to the regression theory, more data can reduce the effect of noise and reveal a pattern closer to the true pattern [7]. Hence, using more than two training inputs in our analysis may produce a better prediction because it reduces the noise from imprecise reuse distance measurement and reference histogram construction.

The extension is straightforward. For each input, we have an equation as shown in (3). For each bin, instead of two linear equations for two unknowns, we have as many equations as the number of training runs. We use the *least square regression* [7] to determine the best values for the two unknowns. We explored regression on three to six training inputs in our previous experiments [5]. In this paper, four training inputs are used in regression. Although more training data can lead to better results, they also lengthen the profiling process.

$$d_i = c_i + e_i * f_i(s). \tag{3}$$

2.1.4 Recording the Overall Model

We select large enough sizes of profiling inputs so that it is unlikely that references having different patterns will 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 Fig. 4.

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. We predict the reuse distance for each group g_i via their particular formula, as shown by (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 estimate of the miss rate. The detailed algorithms for reuse distance prediction and cache miss rate estimation are described in Fig. 5.

```
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 formula in model.formulas do
      f = formula.pattern;
      distance = formula.c + f($size_d$) * formula.e;
      rd[index]=distance;   index++;
    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] $\geq$ $size_c$) missed ++;
    end for
    missRate = missed / model.groupNo;
  end
end algorithm
```

Fig. 5. Cache miss rate estimation.

From our 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 a shorter reuse distance will hit in cache because of temporal locality. The second property is that reuse distances of references having linear or sublinear 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

```
algorithm MaxMR(model, $size_c$,criticalGroup)
  input: model: the locality model
         $size_c$: the cache size
         criticalGroup: the non-constant group
                        with the shortest distance in model
  output: maxMR: maximum miss rate
          $T_s$: threshold data input size
  begin
    missedGroupNo = 0;
    foreach formula in model.formulas do
      if ((formula.pattern != $p_{const}$) ||
                        (formula.c $\geq$ $size_c$))
        missedGroupNo++;
    end foreach
    maxMR = missedGroupNo / model.groupNo;
    if (criticalGroup != null )
      $T_s$ = GetCriticalInputSize(criticalGroup,$size_c$);
    else $T_s$ = null;
  end
end algorithm

subroutine GetCriticalInputSize(formula, distance)
  input: formula: the formula of the given group
         distance: the threshold reuse distance
  output: $T_s$: threshold data input size
  begin
    f = formula.pattern;
    $T_s = f^{-1}((distance - formula.c)/formula.e)$;
  end
end subroutine GetCriticalInputSize
```

Fig. 6. Max miss rate and threshold input prediction.

nonconstant pattern have a reuse distance larger than the cache size.

Fig. 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 ($G = 1,000$ 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 the cache miss rate reaches the maximum value for the program and the given cache configuration. To estimate its value, we only need to consider the single nonconstant 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, $d_j \geq size_c$. By manipulating (3), we can directly calculate the required input data size from this condition:

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

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 3.2).

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 pairwise 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 Accuracy of Miss Rate Prediction

3.1 Methodology

We use the cache simulator *Cheetah* [8] included in the SimpleScalar 3.0 toolset [9] 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* [6] 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 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 percent of the accesses and account for less than 10 percent of the misses.

To evaluate our prediction, we postprocess 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 belong to *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.

Table 1 lists tested benchmarks from the Spec95, Spec2K, NAS, and Olden benchmark suites. A program must take inputs of many different sizes for evaluation. The reuse behaviors for other Spec programs have been studied [3]. Their pattern is as predictable as the ones shown next. For example, the miss rate prediction for *Go* and *Vortex* from Spec95 would be a constant line similar to *Gcc*, which we will examine in detail. The rest of the Olden programs are not included because it is difficult to construct multiple inputs with varying data sizes. Nevertheless, the current set of programs represent common data access patterns: *Applu*, *Swim*, *Tomcatv*, *SP*, *FFT*, and *ADI* represent scientific programs operating on array data, and *Gcc*, *Li*, *BH*, and *EM3D* represent integer programs operating on pointer data.

For each test program, the third column of Table 1 shows the main patterns identified for each of the programs. The

TABLE 1
Cache Block Reuse Pattern Prediction

Bench-mark	Description	Patterns	Input	Data size in cache blocks	Memory references	Accuracy (%)
Applu	solution of five	const	ref(60^3)	5.70M	4.23B	**96.1**
(Spec2K)	coupled nonlinear	3rd roots	train(24^3)	333K	230M	**97.0**
	PDE's	linear	test(12^3)	34.5K	22.3M	
Swim	finite difference	const	ref(512^2)	461K	132M	**98.9**
(Spec95)	approximations for	2nd root	400^2	289K	80.7M	**98.7**
	shallow water equation	linear	200^2	74.3K	20.2M	
SP	computational fluid	const	50^3	1.21M	443M	**96.5**
(NAS)	dynamics (CFD)	3rd roots	32^3	323K	110M	**97.0**
	simulation	linear	28^3	218K	74.4M	
Tomcatv	vectorized mesh	const	ref(513^2)	459K	39.4M	**94.3**
(Spec95)	generation	2nd root	train(257^2)	116K	9.83M	
		linear	400^2	282K	39.1M	**94.6**
FFT	fast Fourier	const	512^2	262K	67.8M	**91.8**
	transformation	2nd root	256^2	65.6K	15.3M	**94.0**
		linear	128^2	16.5K	3.45M	
Gcc	based on the		cp-decl	86.6K	134M	**98.7**
(Spec95)	GNU C compiler	const	amptjp	53.4K	103M	**98.7**
	version 2.5.3		genoutput	18.2K	9.71M	
Li		const	combination-all	45.4K	114M	**94.5**
(Spec95)	Xlisp interpreter	2nd root	combination-prefix-L	18.5K	20.3M	**89.8**
		linear	simplify	4.55K	1.10M	
ADI	two-dimensional	const	500^2	188K	5.99M	**97.5**
	alternate direction	2nd root	300^2	67.5K	2.16M	**94.3**
	implicit integration	linear	200^2	30.0K	959K	
BH	Barnes & Hut N-body	const	16384	114K	2.71B	**99.5**
(Olden)	Force Computation	2nd root	8192	57.7K	1.18B	**99.7**
		linear	4096	28.8K	509M	
EM3D	Electromagnetic wave	const	400 50 75	35.8K	3.8M	**97.7**
(Olden)	propagation in a	3rd roots	200 50 75	18.0K	2.0M	**98.0**
	3D object	linear	100 50 75	9K	1.1M	
					Average	**96.4**

table also shows the accuracy of our prediction scheme for reuse distance. Let x_i and y_i be the size of the ith 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 percent. We use $1 - E/2$ as the accuracy. It measures the overlapping parts of the two histograms, ranging from 0 percent (no match) to 100 percent (perfect match). To show the accuracy of our prediction of reuse distance, we choose three inputs for each benchmark, as listed in Column 4. Columns 5 and 6 summarize these inputs by data input size in cache blocks and the total number 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 last column.

We use original or modified inputs from Spec95/2K for all Spec benchmarks except for *Li*. The *ref* input of *Spec95/Li* contains a set of lisp programs that need to be loaded one by one in a certain order. Since more than three inputs are necessary for evaluation and regression-based pattern recognition, we collected the lisp programs from [10] and composed different combinations of these programs as additional inputs. In Table 1, input *combination-all* includes a total of 54 lisp programs while input *combination-prefix-L* includes eight programs with a name starting with letter *l*.

3.2 Accuracy Across Program Inputs

3.2.1 Fully Associative Cache Results

Shown in Figs. 7, 8, and 9 are the predicted and measured cache miss curves for each of the 10 applications, assuming a fully associative cache. In each pairing, the upper graph is the estimate for a 64 KB (2K blocks) cache and the lower graph is the estimate for a 1 MB (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 in $\log$ scale except for the ones for *Gcc*.

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 64 KB cache. At input data sizes that achieve the maximum miss rate, all hits in the cache are due to the portion of the reuses with a constant reuse distance within the cache size.

Each graph shows four curves for the given benchmark and cache size: three estimated based on different training inputs and one measured with the Cheetah simulator [8].

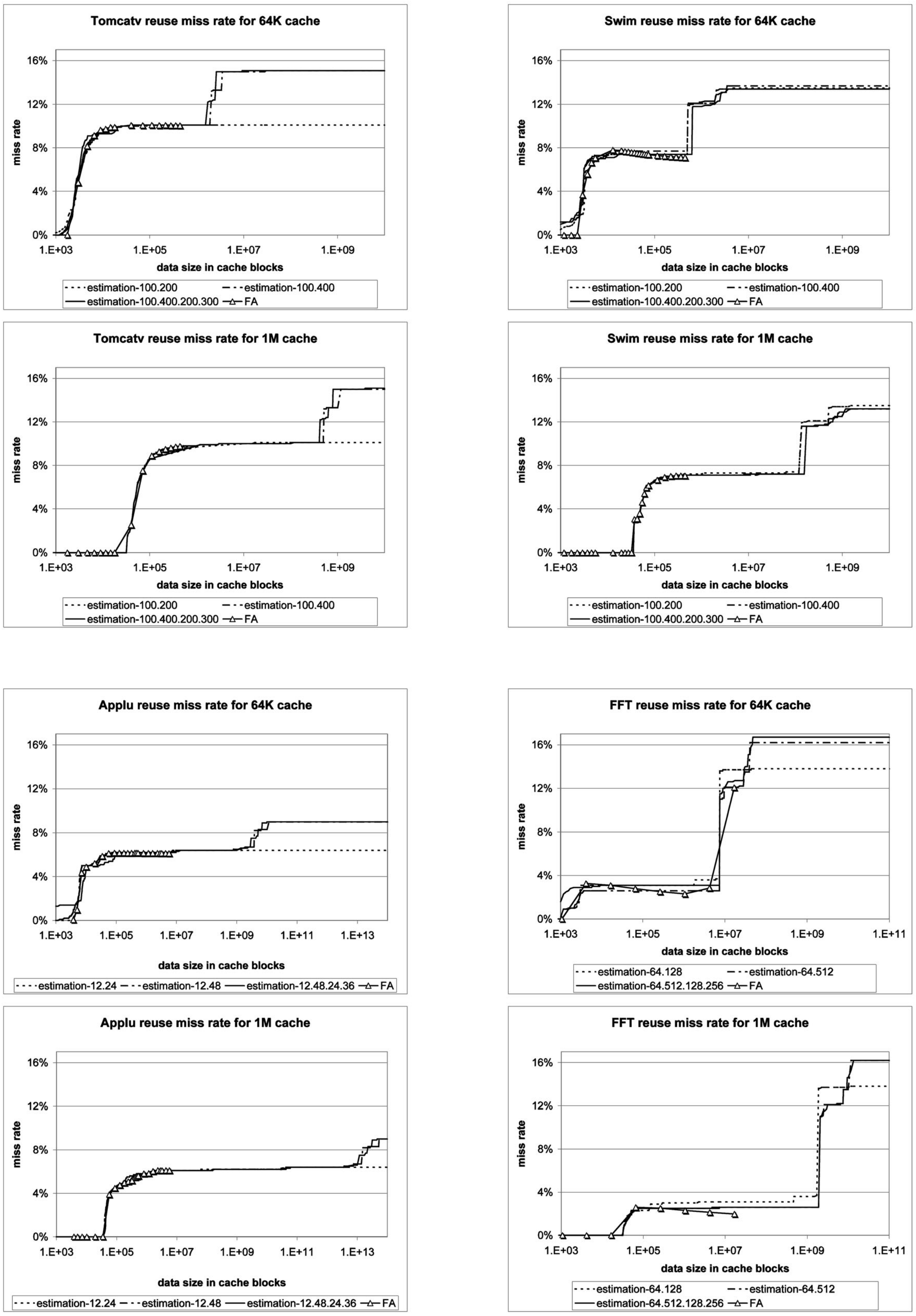


Fig. 7. Estimated miss rates for *Tomactv*, *Swim*, *Applu*, *FFT*, 64 KB and 1 MB caches.

We pick a set of four training inputs for every benchmark and make three different predictions: The first is from the two smallest inputs, the second from the smallest and the largest inputs, and the last one from regression analysis on all four inputs. To compare their accuracy, a fourth curve is added to show the measured miss rate for the fully associative cache. Figs. 7, 8, and 9 show the three prediction curves overlap or differ very little in the early part, but

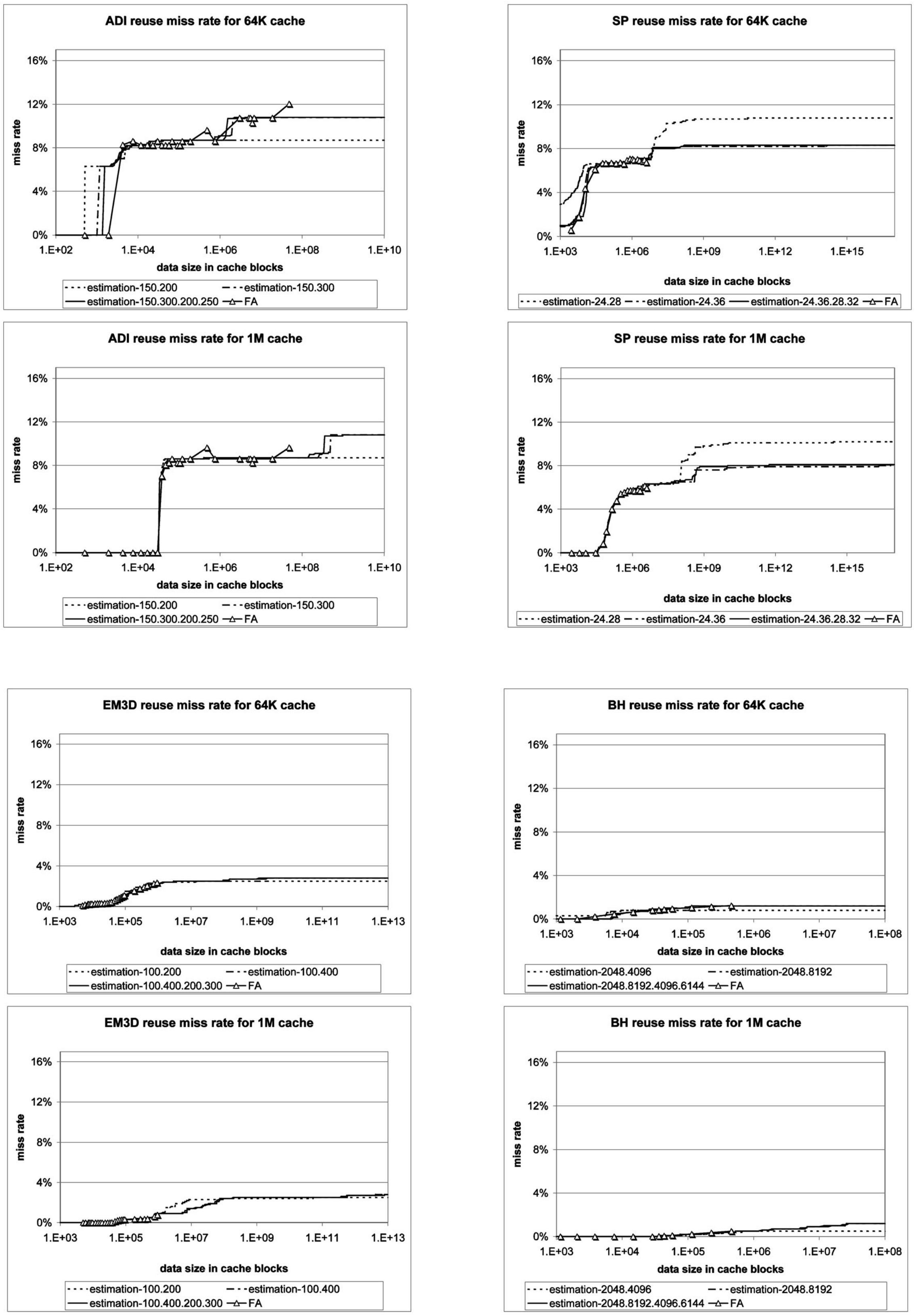


Fig. 8. Estimated miss rates for *ADI*, *SP*, *EM3D*, *BH*, 64 KB and 1 MB caches.

diverge when the input size becomes very large. Unfortunately, the divergence points are much beyond the data size that we can simulate and check. Note that the x-axis is in a logarithmic scale.

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

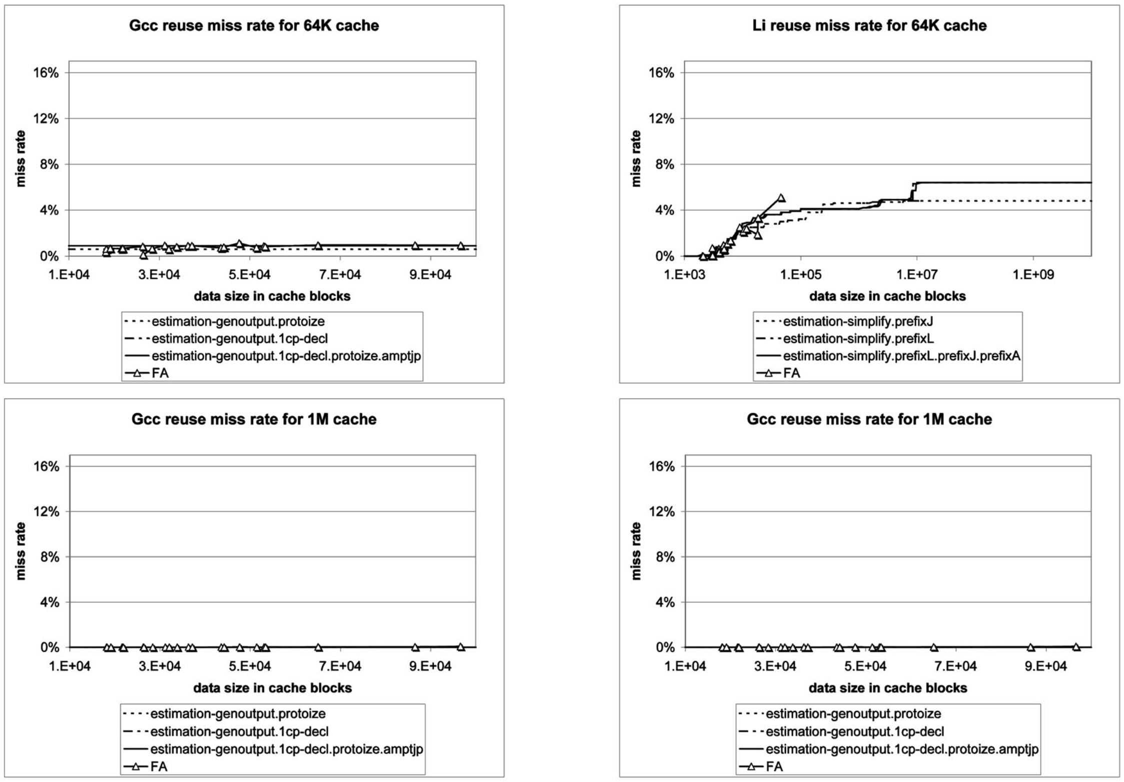


Fig. 9. Estimated miss rates for *Gcc and Li*, 64 KB and 1 MB caches.

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 models. The predictions from the smallest and the largest inputs and from the regression analysis are more accurate than those from the two smallest inputs.

Gcc is notable in its lack of features. The constant pattern has the best fit. In *Gcc*, 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 horizontal lines for both the 64 KB and 1 MB caches (the lines are on the zero mark for a 1 MB cache).

3.2.2 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 Figs. 10, 11, and 12, 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. The predicted curve is the same as the regression-based prediction shown in Figs. 7, 8, and 9.

Overall, it is difficult to separate the different lines in each of the plots because the curves across different configurations are so similar. However, this also demonstrates that our modeling technique makes accurate predictions. Fig. 13 summarizes the *relative hit rate* error across all the applications and cache configurations. We use the relative error for hit rate instead of miss rate as the measure because hit rate is more stable than the miss rate in regions where the miss rate is near zero.

From Fig. 13, the average relative error is less than 1 percent between the model and a simulated fully associative cache (FA). The error is below 2 percent for all applications when the cache associativity is 4-way or greater. Two applications, *Tomcatv* and *Swim*, have a higher relative error (max 7.9 percent) when the associativity is small. Referring back to Fig. 10, the direct mapped cache plots 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 that, in many applications, the miss rate is dominated by capacity misses [11]. Similarly, the error in absolute miss rate is less than 1 percent for fully associative cache and always below 2 percent when the cache associativity is 4-way or greater.

The prediction method does not distinguish program inputs that have the same data size. Different layouts of the same data may lead to very different cache performance. In fact, our technique cannot predict or even measure the quality of the data layout in a program. The reason that the prediction results are accurate is that the quality of the data layout does not change significantly across different inputs of a program. This is true for both the scientific programs and pointer-intensive programs we tested.

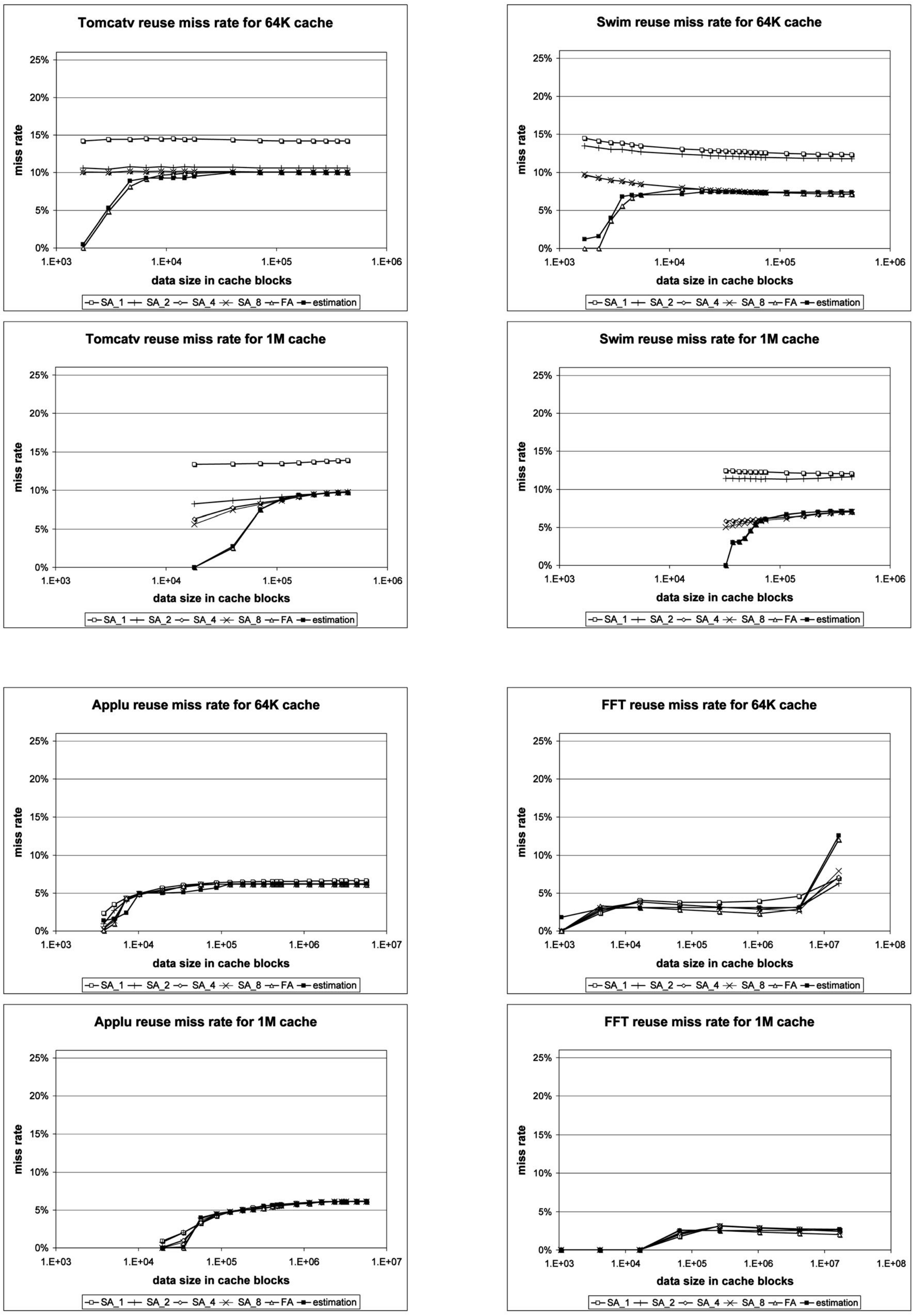


Fig. 10. Verification miss rate estimates *Tomactv*, *Swim*, *Applu*, *FFT*, 64 KB and 1 MB caches.

The program *ADI* exhibits interesting behavior in Fig. 11. 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 at which the cache's 1 MB capacity is exceeded. However, the direct mapped cache's miss rate exhibits a slower rise between the

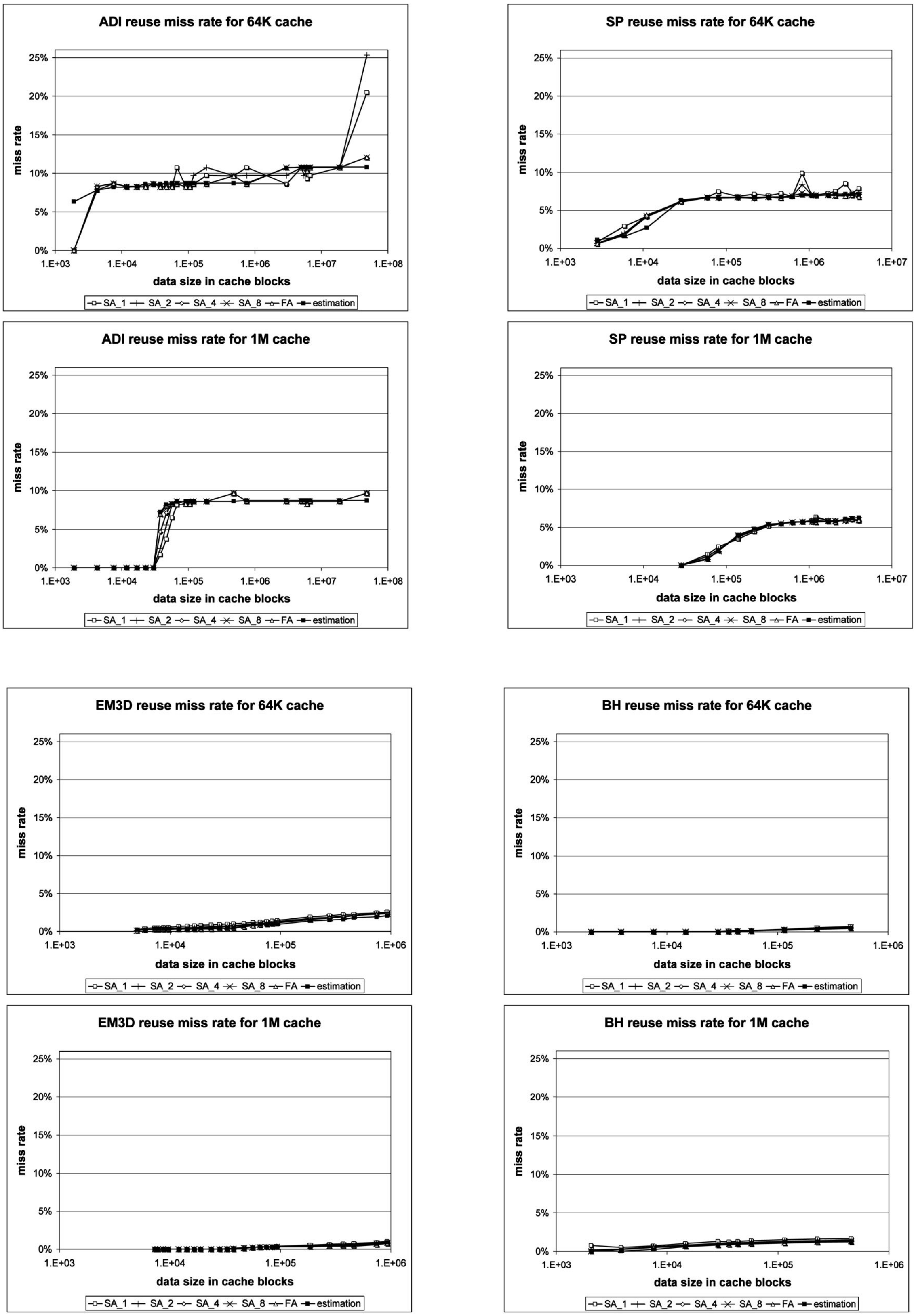


Fig. 11. Verification miss rate estimates *ADI*, *SP*, *EM3D*, *BH*, 64 KB and 1 MB caches.

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

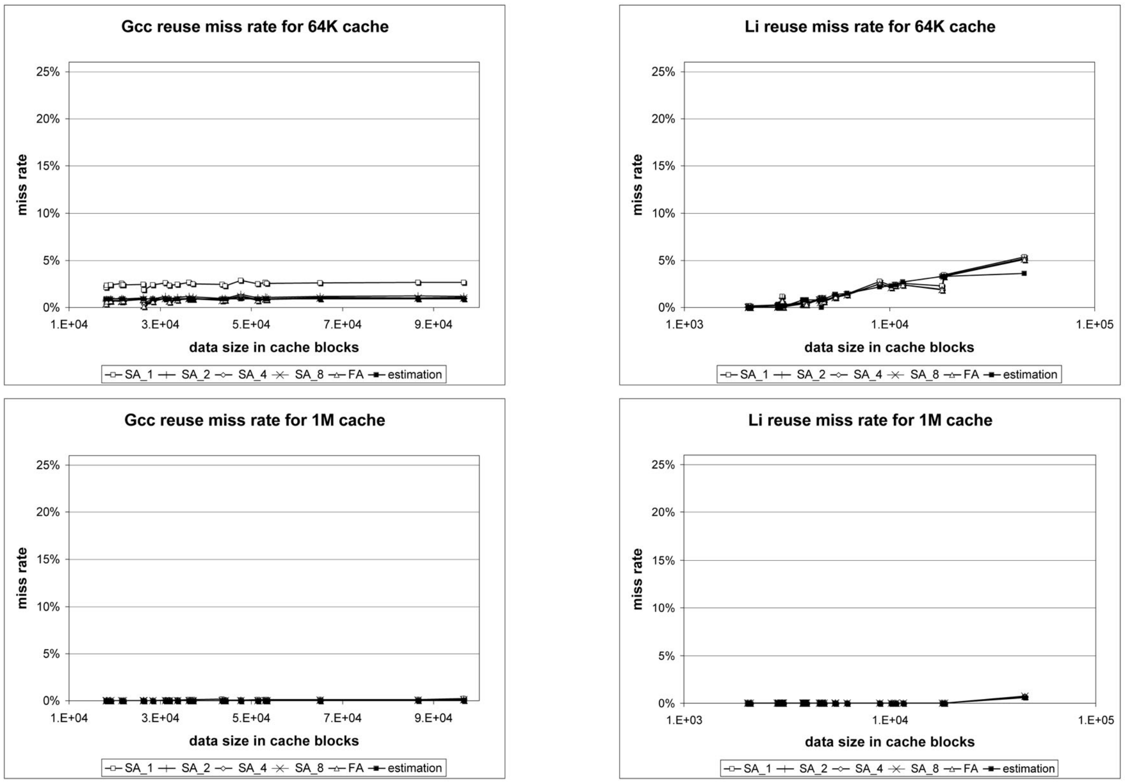


Fig. 12. Verification miss rate estimates *Gcc and Li*, 64 KB and 1 MB caches.

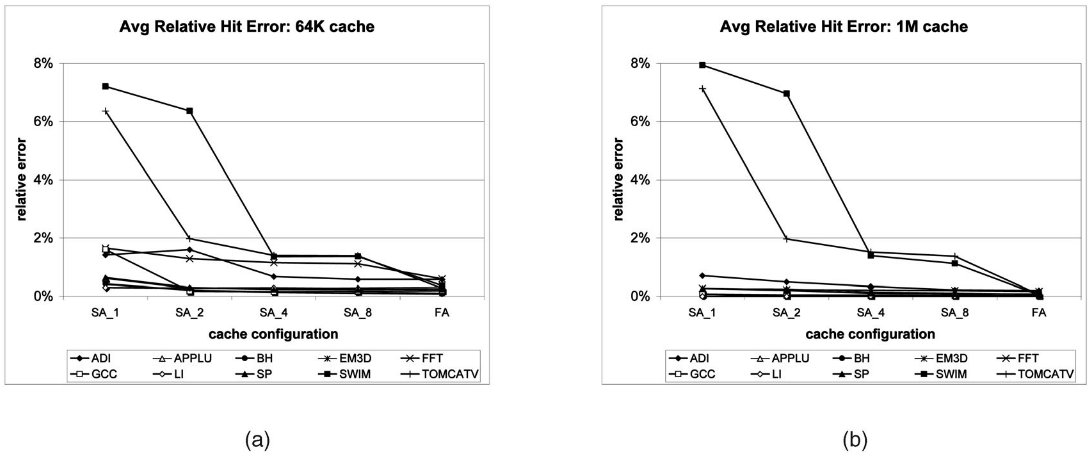


Fig. 13. Relative hit rate error. (a) 64 KB cache. (b) 1 MB cache.

randomly relative to their access order, which happens to improve the miss rate overall for a limited data set range.

3.3 Accuracy Across Cache Configurations

We now measure the prediction accuracy across various cache sizes. We pick three programs, *BH*, *EM3D*, and *Swim*, that are representative and omit the rest. Fig. 14 shows the comparisons of cache miss rate predictions to actual miss rates when the size of the cache varies. The data size we use for this comparison is the one that has the **largest error** of those in Figs. 10 and 11.

The data points shown in Fig. 14 are purely predictions using models generated in the prior section without running the program inputs, let alone simulating them on different cache sizes. Yet, most predictions match the simulation results closely, as shown by the first two graphs. What is remarkable is the high accuracy even for small cache sizes, starting from k blocks for k-way associative cache, where k is between 1 and 8. Not all predictions are as accurate. We pick the worst case—the second smallest test input for *Swim*—and show its poor prediction accuracy for direct-mapped and set-associative caches in the third graph. However, this is an extreme case. Most predictions are accurate, similarly to those shown in the first two graphs.

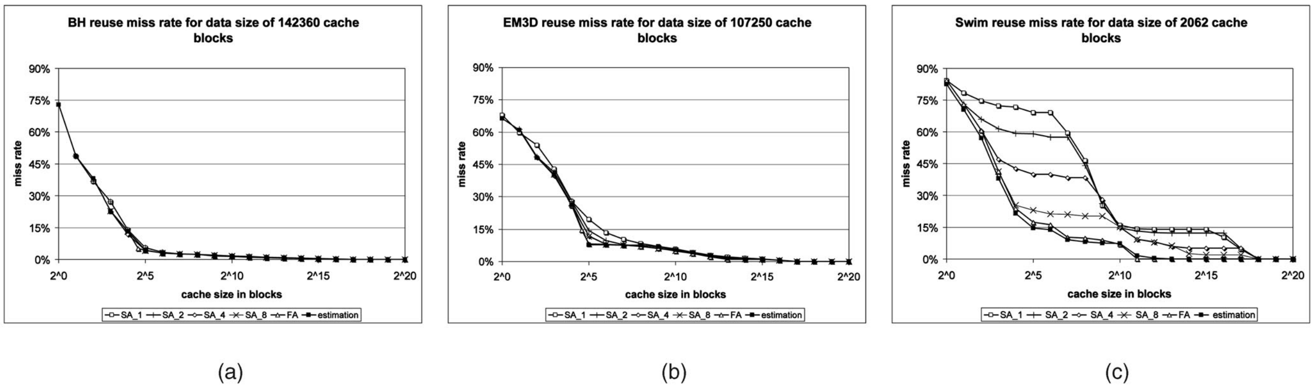


Fig. 14. Verification miss rate estimates for (a) *BH*, (b) *EM3D*, and (c) *Swim* with constant data size on direct-map, two-way, four-way, eight-way, and fully associative caches of all sizes.

4 Cache Miss Rate Visualization and Uses

Since two-dimensional graphs cannot show miss rates for all program inputs and cache configurations at the same time, this section presents a Web-based tool that displays miss rates as a three-dimensional plot. We use this tool to evaluate a compiler and discuss other uses in machine and benchmark-set design.

4.1 A Visualization Tool

The interactive tool displays the miss rates of a program across program inputs and cache configurations. By varying the cache size (x-axis) and the data size (y-axis), the tool generates a 3D graph of the miss rates covering a broad region of interest. The miss rate surface is formed by connecting every three adjacent points (along the x and y directions) to approximate, via linear interpolation, the miss rates over the region. While a 3D graphing aids in quick exploration over a large combination of cache and data sizes, two-dimensional graphs can also be generated for a specific cache or data set size. Appendix I, which can be found on the Computer Society Digital Library at http://computer.org/tc/archives.htm, gives examples of both 3D and 2D outputs of our tool as well as a detailed description of the tool interface.

The visualization tool can be accessed via the Internet at http://www.cs.rochester.edu/research/locality with any browser equipped with Java 1.4 JVM and Java 3D. The system currently runs on practically all x86 machines (Windows, Linux), Sun, SGI, and IBM platforms.

4.2 Compiler Evaluation

Typically, compiler transformations are evaluated by testing programs for a few inputs on a few machines. While running on real systems allows accurate measurement of a complete system, the few results cannot reliably predict the effect on other program inputs and on other machines. By using miss-rate prediction, a compiler writer can examine memory behavior across all program inputs and all cache sizes. The visualization tool is a convenient method for quickly summarizing the results to see if a certain compiler transformation will provide better performance on a certain memory hierarchy.

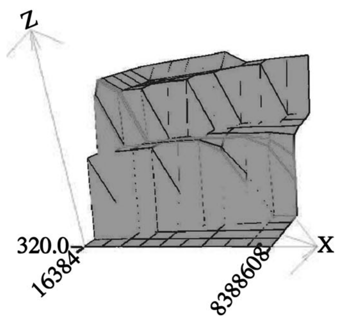
To demonstrate this use, we use the visualization tool to evaluate two aggressive compiler transformations: reuse-based loop fusion and data regrouping [12]. Ding and Kennedy showed that these transformations recombined all loops and reshuffled all arrays and, as a result, reduced the number of cache misses by 50 percent on average. A similar fusion method was later used in the Intel Itanium compiler and helped to improve the 14 Spec2K FP benchmarks by, on average, 12 percent [13]. Both studies used a single input on a single machine. The graphs in Fig. 15 show the effect of the transformations on *Swim* for a range of inputs and cache sizes. The base version is generated by the Alpha compiler using the highest optimization level (*-O5*). The other two are transformed versions by reuse-based loop fusion and by a combination of loop fusion and data regrouping.

The graphs in Fig. 15 are scaled identically on all axes to simplify comparisons (lower is better). The different shapes in the plots show the improvement from the transformations is not uniform for *Swim*. The improvement is greater for larger data inputs and the difference is greater for the combined transformation than for loop fusion alone. A similar comparison of *Tomcatv* is included in Appendix II, which can be found on the Computer Society Digital Library at http://computer.org/tc/archives.htm. Since the data in Fig. 15 are predictions, their accuracy was evaluated using the same techniques as discussed in Section 3 and found to be accurate. Only with the miss-rate prediction and the visualization tool could we, for the first time, fully observe the input and machine-dependent effect of compiler transformations.

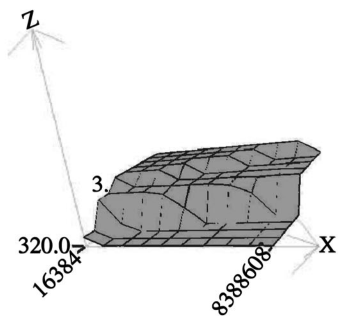
Although the tool helps to evaluate a compiler transformation, the average miss rate itself is not specific enough to drive compiler transformations. Still, similar prediction techniques can be used at a finer granularity. Fang et al. found an accurate miss-rate prediction for most memory references in SPEC2Kfp programs [14]. Such prediction is directly useful to a compiler.

4.3 Other Uses of Miss-Rate Prediction

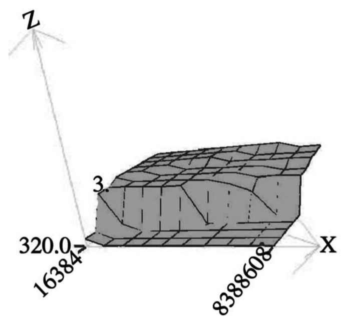
By predicting complete program and machine behavior, the tool can help to build cost-effective computer systems and to design better benchmark suites.



(a)



(b)



(c)

Fig. 15. Three-dimensional plots of unoptimized and optimized Swim. (a) Original Swim. (b) Fused Swim. (c) Fused Grouped Swim.

TABLE 2
Applu and *Swim*: Mini-Ref Inputs

Benchmark	Input	Data size in cache blocks	Memory references	64KB 2-way SA cache miss rate	1MB 4-way SA cache miss rate
Applu (Spec2K)	ref	5.70M	4.23B	6.22%	6.11%
	mini-ref	470K	308M	6.19%	5.52%
	$\frac{ref}{miniref}$	12.1	13.7	1.00	1.11
Swim (Spec95)	ref	461K	132M	11.7%	6.87%
	mini-ref	115K	31.6M	11.9%	6.29%
	$\frac{ref}{miniref}$	4.01	4.18	0.98	1.09

4.3.1 Cost-Effective Memory Hierarchy Design

Today's computing centers often use thousands of processors to run a few large applications. Rather than having to simulate numerous inputs and extrapolating how the system would perform from those runs, designers could save both time and money by being able to see the change in miss rates as cache size changes. Additional uses include:

1. allowing customers to quickly determine a balance between price and performance for memory configurations,
2. evaluating the cache performance of data sets too large to simulate on any existing machine,
3. providing insight into existing systems used to execute new or larger applications, and
4. determining whether a new system upgrade can provide improved use of the memory hierarchy.

4.3.2 Benchmark Set Design

With accurate estimates of cache performance across numerous inputs, benchmark design may be improved to allow faster evaluation of systems and optimizations. By building benchmarks that run on values directly after knees in the data size versus miss rate plots, the smallest possible data size for a given miss rate could be used in testing. For two SPEC programs, Table 2 shows the smallest data input size (labeled mini-ref) that gives a similar cache miss rate for 64 KB and 1 MB cache as the reference input does. We obtained the new input size from the miss-rate prediction from Fig. 7 without testing any additional inputs. The new input is a factor of 4 or 12 smaller, but its miss rate differs by no more than 0.6 percent from the larger input. A smaller input saves time in testing. The right input size depends on the program as well as the cache configuration. The prediction tool helps a user to quickly find the smallest input size that yields a particular cache miss rate.

Eeckhout et al. studied correlations between miss rates of nine programs across 79 inputs using principal components analysis followed by hierarchical clustering [15]. 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. Furthermore, the suggestion is parameterized and, therefore, tailored to any cache configuration being considered.

5 Related Work

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

Dependence analysis analyzes data reuses in loop nests and can estimate the number of capacity misses in scientific programs [16], [17], [18], [19]. Other researchers used various types of array sections to measure data access in loops and procedures [20], [21], [22], [23], [24]. One limitation of dependence analysis is that it does not model cache interference caused by the data layout. An early technique used efficient heuristics [25]. Recent studies used precise (though worst-case superexponential) methods [26], [27], [28] or their fast approximation through sampling [29],

[30]. These methods are more powerful, but they are limited to programs written in loop nests with regular array subscripts. For full applications, researchers have combined compiler analysis with cache simulation to measure reference locality within and between loop nests [11] and fine-grained reuse and program balance [31].

For regular loop nests, compiler analysis identifies not only the cache behavior but also its exact causes in the code. However, 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 [32] and cache reuse [33], [34], to evaluate the effect of program transformations [4], [33], [34], [35], and to annotate programs with cache hints to a processor [36]. The last work used reuse distance profiles to generate hints in SPEC95 FP benchmarks and improved performance by 7 percent on an Itanium processor [36]. The techniques in this paper will allow compiler writers to estimate cache behavior for data inputs based on a few profiling runs.

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 [1]. It was later extended to measure the miss rate in direct-mapped caches and practical set-associative caches [2]. 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.

6 Summary

This paper presents an algorithm for estimating the cache 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 that the prediction accuracy is always higher than 99 percent for fully associative caches and better than 98 percent for caches of limited associativity for all but two programs, 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. Building on the new model, the paper presents an interactive visualization tool that shows miss rates across all program inputs and cache sizes. Since memory hierarchy performance increasingly determines the performance, cost, and energy efficiency of modern computer systems, the tool should help to improve program, machine, and benchmark-set design and implementation.

Acknowledgments

The authors would like to thank the the anonymous reviewers of the *IEEE Transactions on Computers* for their insightful comments and helpful suggestions. The work and its presentation also benefited from discussions with their colleagues at the University of Rochester as well as the anonymous reviewers of the PACT 2003 conference. This work is supported by the US National Science Foundation (Contract Nos. CCR-0238176, CCR-0219848, and EIA-0080124), the US Department of Energy (Contract No. DE-FG02-02ER25525), and an equipment grant from IBM.

References

[1] R.L. Mattson, J. Gecsei, D. Slutz, and I.L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems J.*, vol. 9, no. 2, pp. 78-117, 1970.

[2] M.D. Hill, "Aspects of Cache Memory and Instruction Buffer Performance," PhD thesis, Univ. of California, Berkeley, Nov. 1987.

[3] C. Ding and Y. Zhong, "Predicting Whole-Program Locality with Reuse Distance Analysis," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* June 2003.

[4] K. Beyls and E. D'Hollander, "Reuse Distance as a Metric for Cache Behavior," *Proc. IASTED Conf. Parallel and Distributed Computing and Systems,* Aug. 2001.

[5] X. Shen, Y. Zhong, and C. Ding, "Regression-Based Multi-Model Prediction of Data Reuse Signature," *Proc. Fourth Ann. Symp. Los Alamos Computer Science Inst.*, Nov. 2003.

[6] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, June 1994.

[7] J.O. Rawlings, *Applied Regression Analysis: A Research Tool.* Wadsworth and Brooks, 1988.

[8] R.A. Sugumar and S.G. Abraham, "Efficient Simulation of Multiple Cache Configurations Using Binomial Trees," Technical Report CSE-TR-111-91, Computer Science and Eng. Division, Univ. of Michigan, 1991.

[9] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," Technical Report CS-TR-97-1342, Dept, of Computer Science, Univ. of Wisconsin, June 1997.

[10] K.D. Forbus and J. de Kleer, *Building Problem Solvers.* MIT Press, 1993.

[11] K.S. McKinley and O. Temam, "Quantifying Loop Nest Locality Using SPEC'95 and the Perfect Benchmarks," *ACM Trans. Computer Systems,* vol. 17, pp. 288-336, Nov. 1999.

[12] C. Ding and K. Kennedy, "Improving Effective Bandwidth through Compiler Enhancement of Global Cache Reuse," *Proc. Int'l Parallel and Distributed Processing Symp.*, Apr. 2001, http://www.ipdps.org.

[13] J. Ng, D. Kulkarni, W. Li, R. Cox, and S. Bobholz, "Inter-Procedural Loop Fusion, Array Contraction and Rotation," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* Sept. 2003.

[14] C. Fang, S. Carr, S. Onder, and Z. Wang, "Reuse-Distance-Based Miss-Rate Prediction on a per Instruction Basis," *Proc. First ACM SIGPLAN Workshop Memory System Performance,* June 2004.

[15] L. Eeckhout, H. Vandierendonck, and K.D. Bosschere, "Workload Design: Selecting Representative Program-Input Pairs," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* Sept. 2002.

[16] C. Cascaval and D.A. Padua, "Estimating Cache Misses and Locality Using Stack Distances," *Proc. Int'l Conf. Supercomputing,* June 2003.

[17] T. Fahringer, "Estimating Cache Performance for Sequential and Data Parallel Programs," *Proc. Int'l Conf. High Performance Ccomputing and Networking,* 1997.

[18] K. Gallivan, W. Jalby, and D. Gannon, "On the Problem of Optimizing Data Transfers for Complex Memory Systems," *Proc. Second Int'l Conf. Supercomputing,* July 1988.

[19] A. Porterfield, "Software Methods for Improvement of Cache Performance," PhD thesis, Dept. of Computer Science, Rice Univ., May 1989.

[20] M. Burke and R. Cytron, "Interprocedural Dependence Analysis and Parallelization," *Proc. SIGPLAN '86 Symp. Compiler Construction,* June 1986.

[21] D. Callahan, J. Cocke, and K. Kennedy, "Analysis of Interprocedural Side Effects in a Parallel Programming Environment," *J. Parallel and Distributed Computing,* vol. 5, pp. 517-550, Oct. 1988.

[22] P. Havlak and K. Kennedy, "An Implementation of Interprocedural Bounded Regular Section Analysis," *IEEE Trans. Parallel and Distributed Systems,* vol. 2, pp. 350-360, July 1991.

[23] Z. Li, P. Yew, and C. Zhu, "An Efficient Data Dependence Analysis for Parallelizing Compilers," *IEEE Trans. Parallel and Distributed Systems,* vol. 1, pp. 26-34, Jan. 1990.

[24] R. Triolet, F. Irigoin, and P. Feautrier, "Direct Parallelization of CALL Statements," *Proc. SIGPLAN '86 Symp. Compiler Construction,* June 1986.

[25] J. Ferrante, V. Sarkar, and W. Thrash, "On Estimating and Enhancing Cache Effectiveness," *Proc. Fourth Int'l Workshop Languages and Compilers for Parallel Computing* U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, eds., Aug. 1991.

[26] K. Beyls, "Software Methods to Improve Data Locality and Cache Behavior," PhD thesis, Ghent Univ., 2004.

[27] S. Chatterjee, E. Parker, P.J. Hanlon, and A.R. Lebeck, "Exact Analysis of the Cache Behavior of Nested Loops," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* 2001.

[28] S. Ghosh, M. Martonosi, and S. Malik, "Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior," *ACM Trans. Programming Languages and Systems,* vol. 21, no. 4, 1999.

[29] X. Vera, N. Bernudo, J. Llosa, and A. Gonzalez, "A Fast and Accurate Framework to Analyze and Optimize Cache Memory Behavior," *ACM Trans. Programming Languages and Systems,* vol. 26, Mar. 2004.

[30] J. Xue and X. Vera, "Efficient and Accurate Analytical Modeling of Whole-Program Data Cache Behavior," *IEEE Trans. Computers,* vol. 53, no. 5, May 2004.

[31] J. Mellor-Crummey, R. Fowler, and D.B. Whalley, "Tools for Application-Oriented Performance Tuning," *Proc. 15th ACM Int'l Conf. Supercomputing,* June 2001.

[32] Z. Li, J. Gu, and G. Lee, "An Evaluation of the Potential Benefits of Register Allocation for Array References," *Proc. Workshop Interaction between Compilers and Computer Architectures in Conjunction with the HPCA-2,* Feb. 1996.

[33] C. Ding, "Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse," PhD thesis, Dept. of Computer Science, Rice Univ., Jan. 2000.

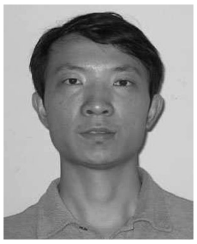
[34] Y. Zhong, C. Ding, and K. Kennedy, "Reuse Distance Analysis for Scientific Programs," *Proc. Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers,* Mar. 2002.

[35] G. Almasi, C. Cascaval, and D. Padua, "Calculating Stack Distances Efficiently," *Proc. First ACM SIGPLAN Workshop Memory System Performance,* June 2002.

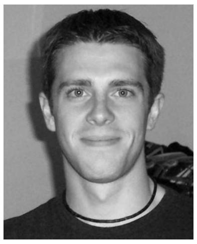
[36] K. Beyls and E. D'Hollander, "Reuse Distance-Based Cache Hint Selection," *Proc. Eighth Int'l Euro-Par Conf.,* Aug. 2002.

Yutao Zhong received the BS and ME degrees in computer science from Nanjing University, China, in 1997 and 2000, respectively, and the PhD degree from the Department of Computer Science at the University of Rochester. She is an assistant professor of computer science at George Mason University. Her research focuses on compiler-assisted program locality analysis, including efficient locality pattern modeling and prediction, memory performance estimation, and program data management based on reference affinity. She is a member of the IEEE Computer Society.

Steven G. Dropsho received the PhD degree in computer science from the University of Massachusetts, Amherst, and the BS and MS degrees in electrical engineering from the University of Wisconsin, Madison. He is currently a postdoctoral researcher at the Swiss Institute of Technology (EPFL) in Lausanne, Switzerland. His current research is in high performance, scalable server systems. He is a member of the IEEE.



Xipeng Shen received the PhD degree in computer science from the University of Rochester. He is an assistant professor of computer science at the College of William and Mary. His research focuses on large-scale program behavior analysis, in particular, using statistic techniques to detect and predict program behavior patterns and apply them to program optimizations. His past work covers locality phase analysis, efficient data locality measurement and prediction, and parallel sorting.



Ahren Studer received the BS degree in electrical engineering and computer science from the University of Rochester. He is currently a graduate student in the Department of Electrical and Computer Engineering at Carnegie Mellon University in Pittsburgh, Pennsylvania. His current research is in securing vehicular ad hoc networks and authentication and privacy in social networks.

Chen Ding received the PhD degree from Rice University, the MS degree from Michigan Technological University, and the BS degree from Beijing University. He is an associate professor in the Department of Computer Science at the University of Rochester. He is the recipient of an Early Career Principal Investigator award from the Office of Science of the US Department of Energy, a CAREER award from the US National Science Foundation, and a best-paper award from the IEEE International Parallel and Distributed Processing Symposium. He was the coorganizer of the First and general chair of the Second ACM SIGPLAN Workshop on Memory System Performance (MSP) in 2002 and 2004. He is a member of the IEEE Computer Society.