An Interactive 3-D Visualization Tool for Accurate Estimation of Miss Rates

Ahren Studer, Yutao Zhong, and Chen Ding

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

Abstract

Making use of information on cache performance requires a quick way to comprehend how the miss rate for an application changes as cache and input size varies. In 1970, Mattson et al. showed how to measure miss rates for all cache sizes. Recently, Zhong et al. showed how to predict miss rates for all program input data sizes. This paper builds on the previous results and shows the miss rate of a program as a function over the domain with cache and data input size as two orthogonal dimensions. This paper makes three contributions. First, it presents an interactive tool that visualizes the miss rates in three-dimensional plots. It measures the compounded error of prediction for different cache sizes for program inputs that are never run but along simulated on a cache simulator. Second, it applies predictions to a new set of benchmark programs with dynamic data structures. Finally, it discusses possible uses of the new tool. Experiments show that the compounded prediction error for the hit rate is within 6.5% for caches of all sizes and with a small amount of associativity. The visualization tool can run on any machine with Java 3D. It can be downloaded from http://www.cs.rochester.edu/research/locality.

1 Introduction

As the gap between processor speeds and memory access times grows the efficiency of the memory hierarchy becomes more crucial to overall performance. To achieve a high IPC a program’s reference pattern must match the system’s cache structure. Traditionally most memory hierarchies are static and configured to perform well over a large range of applications. However with large systems purchased solely to run a few applications and the introduction of adaptive architectures, knowledge of cache performance across a large range of both input sizes and cache sizes could be used to drastically improve performance and power savings.

In 1970, Mattson et al. gave an algorithm that measures cache miss rates for all cache sizes for several types of cache including fully associative cache with LRU (least-recently-used) replacement policy [11]. Hill later extended the method to cache with limited associativity [5]. Cache miss rate also depends on the cache block size, although the number of choices is relatively small in practice (five or so cache block sizes versus thousands of cache sizes), allowing for brute force exploration. A remaining problem is the effect of program data inputs: the miss rate from a few runs with specific input sizes yields very little information about future executions on other program inputs.

Recently, Ding and Zhong showed that by analyzing the distance between data reuses, it is possible to predict locality as a function of input data sizes [4]. When a pattern does exist, Zhong et al. showed that it is possible to accurately predict cache miss rates across varying inputs based on two profiling (or training) runs of two different program inputs [16]. For a set of mostly scientific programs, they demonstrated accurate miss rate prediction for all program inputs for 64KB and 1MB cache with 32-byte cache blocks. However,
these results are for these two specific cache sizes and are not sufficient to measure the accuracy of prediction for other cache sizes.

In this paper, we present a new tool for analyzing the cache miss rate as a function over the domain with cache sizes and program input sizes as two orthogonal dimensions. It shows estimated miss rates as a surface in a three-dimensional space. The estimation is not completely accurate. The analysis profiles only two program runs and predicts for all other program inputs. The error is compounded when the prediction is used to estimate the miss rate for different cache sizes. Indeed, the miss rate for most data inputs is estimated without running the program on these inputs let alone simulating these executions on different cache sizes. In this paper, we measure the compounded error of prediction on caches with different associativity and a wide range of sizes from only one cache block to several megabytes.

We apply the analysis to a new set of benchmark programs—Olden benchmark set, whose programs make extensive uses of dynamic data structures. In addition, we discuss several possible uses of the new tool. The tool can run on any machine with Java 3D and can be downloaded from our web site. Equally important, we are currently analyzing another set of benchmark programs from US Department of Energy laboratories and the NAS NPB suite. We expect to add results of these programs into our tool by the end of the summer.

The remainder of this paper is organized as follows. In Section 2 we discuss how the miss rate is estimated, how the 3-d graphs used to represent the miss rate are generated, and the operation of the tool. In Section 3 we discuss how to verify the estimates across both different data sizes and input sizes on a set of dynamic benchmarks. In Section 4 we discuss the results of these comparisons. Possible applications of the tool and related work are discussed in Sections 5 and 6 respectively and in Section 7 we make concluding remarks.

2 Miss Rate Estimation and Interactive Visualization of the Miss Rate

2.1 Miss Rate Estimation

The tool allows a user to quickly view an accurate prediction of the miss rate of a benchmark over a wide range of both input and cache sizes, while requiring only two instrumented executions of the program. By using the approximate reuse distance analysis of Ding and Zhong[4], the reuse histogram can be quickly generated with minimal overhead when compared to older methods of reuse analysis. However, rather than analysis on a per element level we use analysis by cache block. With two reuse distance histograms known, pattern matching can be used to see how reuse distance varies with data size. With this information a model for the benchmark can be generated that can be used to estimate the reuse distance histogram for any execution of the benchmark.

Once the model for a program is defined, given cache size and data size in cache blocks, it is simple to estimate the miss rate [16]. First the data size is used to calculate the new reuse distance histogram for the benchmark from the model. Once the reuse distance histogram is known for the specific instance of the benchmark, the miss rate is found by simply recording the percentage of data references whose reuse distance is greater than the cache size. With this work we focus on extending the estimate across multiple cache and data sizes, rather than multiple data sizes with one or two cache sizes. By estimating across different cache sizes we compound any error due to inaccuracies in the reuse distance histogram generated for a given data size, but the relative error for the hit rate remains low. The estimate is most accurate for a fully associative cache with an LRU replacement scheme since this configuration’s eviction scheme relies on reuse distance. For example assume there exists cache block C and a cache with N blocks. Now if there are N or more distinct cache blocks accessed between two references to C; C will have a reuse distance of N or greater and suffer a capacity miss since it will be replaced prior to the occurrence of the second access. This example shows one of the major weaknesses with the miss rate estimate since only capacity misses are taken into account, leaving conflict and compulsory misses unaccounted for.
2.2 Interactive Visualization of the Miss Rate

2.2.1 Visualization Capabilities of the Tool

The major contribution of this paper is a tool that allows visualization of the miss rate of an application across a large range of both data and cache sizes. To generate the three dimensional plot a set of vectors is generated that hold the given parameters for the execution of the benchmark and the estimated miss rate. This vector can easily be treated as a point in traditional three dimensional space with cache size in the x direction, data size in the y direction, and miss rate in the z direction. With this set of points, a plane can be generated by connecting three adjacent points, generating an approximation for the miss rates of points that would lie between those estimates. Figure 1 shows examples of the three dimensional graphs generated by our tool.

Figure 2 shows a screen shot of our tool with the user interface. Besides for generating graphs of miss rates as two dimensions vary, the tool can also generate estimations for miss rates of an execution with a cache of a specific size and the input size varying as in Figure 3, execution of a benchmark with constant data size on different cache sizes, or execution with a single data and cache size.
2.2.2 Operation of the Tool

To use the tool all one needs is access to the internet, and a browser equipped with the JVM and Java 3D. [http://www.cs.rochester.edu/research/locality](http://www.cs.rochester.edu/research/locality) gives simple instructions on both how to acquire Java 3D and how to operate the applet. Once the applet is loaded the user can do several things with the plots representing the miss rates. First the user must select a benchmark from the menu. This will generate a series of planes on the right side of the window. Depending on whether the user desires accuracy or speed, using the options menu, the number of points used to generate the three dimensional shape can be altered so that the planes offer an approximation closer to the corresponding estimates. To balance speed and accuracy the user can view the miss rates of a program over a large range of values, and subsequently reduce the range of values, effectively zooming in, increasing the number of points in the set around the critical values while discarding points, and time consuming calculations, for estimates in irrelevant portions of the exploration space. Once the proper range of values has been determined the user can use the buttons on the bottom left hand side of the window to manipulate the three dimensional representation for easier viewing. The user can also manipulate the graph by holding down the left or right mouse buttons and moving the mouse, causing the graph to rotate or translate respectively.

It is possible to generate a two dimensional graph of the miss rate as cache or data size varies and the other value is constant. To do this the user must enter the cache size or data size in the text fields on the left hand side of the window, and place a wild card symbol (*) in the field they desire to vary. Finally, a user can find the miss rate for a specific run of the application with a specific data and cache size by entering the information in the correct text fields and pressing the button on the left side of the window.

3 Methodology

In order to collect cache miss rates we use the Cheetah [13] cache simulator included in the Simplescalar 3.0 suite of tools. For all of the simulations the cache line size is set to 32 bytes and the configuration is either fully associative, 2, 4, or 8 way associative, or direct mapped. To collect the addresses of all the loads and stores in a benchmark we use Atom[12] and feed the resulting addresses to Cheetah. Once these same addresses are translated to cache blocks the reuse pattern is recorded.

Cheetah’s output is the overall miss rate of a benchmark, not just the capacity misses which our model is supposed to compute. As previously discussed compulsory misses are not included in the estimate, requiring...
for Cheetah’s output to be processed prior to comparison to the estimation. All reported miss rates are
the reuse miss rate (miss rate without compulsory misses). In order to calculate the reuse miss rate the
following formula is used where \( N_{\text{Total}} \) is the total number of accesses and \( N_{\text{Compulsory}} \) is the total number
of compulsory misses. \( \text{Reuse Miss Rate} = \frac{\text{Miss Rate} \times N_{\text{Total}} - N_{\text{Compulsory}}}{N_{\text{Total}} - N_{\text{Compulsory}}} \)

Originally we planned on evaluating our method on a series of DOE benchmarks, the rest of the NAS
NPB suite, and all of the Olden suite. We plan to add the DOE benchmarks to our tool and verify the accuracy
of the prediction for them and the rest of the NAS benchmarks by the end of the summer. However, we have
encountered a problem with Cheetah while simulating some of these scientific benchmarks since Cheetah
uses a 4 byte integer to store data and there is often more references than can be recorded in that space.
The remainder of the Olden suite was not included in our results due to no way to change the size of the
data for Power, the data size growing too quickly to gather any useful data for TSP and TreeAdd, a constant
data size for all inputs for Perimeter, and exceptions occurring during execution for the other benchmarks.
Those listed here were recently evaluated and are an extension to the prior work by Zhong, Dropsho, and
Ding[16]. However, this new set of benchmarks is the first attempt to apply this technique to benchmarks
with dynamic data structures. This work also extends the evaluation of our method by evaluating how the
effect of our estimate is compounded by estimating the miss rates across a wide range of cache sizes(32 B
to 32 MB). Table 1 lists all of the benchmarks we test. The first and second columns give the name of the
benchmark and a short description. The third column gives the inputs used for generating the model. It is
important to note that we limit the first input for Health to 5 since any other value would result in a different
access pattern during program execution. This is similar to when we use our tool on two or three dimensional
data and we only alter one dimension of the shape to generate and test a model. The fourth column gives
the size of those inputs in terms of cache blocks. The fifth column gives the number of memory references.
The sixth column gives the relative error of the hit rate when compared to the estimate for a 64 kB cache. In
Table 2 we list all of the benchmarks one can view from our tool on the web, including the amount of error
we have found for the estimates.

4 Results

As previously discussed the miss rate estimation is based on a fully associative cache with a LRU replace-
ment policy. However, this type of cache is too slow to implement in higher levels of the memory hierarchy.
For this reason set associative or direct mapped caches are often used. Figure 4 shows the comparison of
cache miss rate predictions to actual cache miss rates when only the data size is allowed to vary. For each
benchmark we evaluate the estimate for both a 64K and 1MB cache. The range of values we sample for
each benchmark is limited by Cheetah’s overhead. Due to this limitation only the results for EM3D allow
for verification of the estimation near knees in the miss rate graph. For the other benchmarks the range of
values occurs on a plateau of the miss rate graph. Figure 5 shows the average relative hit rate error for each
benchmark as associativity varies in the comparisons in Figure 4.

The graphs show that for BH and EM3D the actual miss rates of a fully associative cache follow the
pattern of the estimate, and are accurately predicted using the model, resulting in a maximum average
relative error for the hit rate less than 1%. For HEALTH the estimate does not provide as accurate of a
prediction (3.3% average relative error for hit rates), but this may be due to the large variance in cache
behavior, which is the result of a pattern our model may not have been able to fully capture. Another reason
HEALTH has a large relative error is that with a smaller hit rate a difference of one percent between the
estimate and the actual miss rate results in a larger relative error. For BH and EM3D the hit rates are greater
than 98%, but for HEALTH the hit rate lies between sixty and seventy percent, generating a less stable
relative error.

---

1Equation comes from [16](PACT’03)
Figure 4: Miss rates comparison for BH, EM3D, HEALTH, 64KB and 1MB caches
Table 1: Benchmarks Tested

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Input</th>
<th>Data Size in Cache Blocks</th>
<th>Memory references</th>
<th>Relative Error for estimate with FA 64k cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>BH</td>
<td>Barnes &amp; Hut N-body</td>
<td>4096</td>
<td>2882</td>
<td>509095520</td>
<td>used for model</td>
</tr>
<tr>
<td>(Olden)</td>
<td>Force Computation</td>
<td>8192</td>
<td>5774</td>
<td>117940512</td>
<td>0.049%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12288</td>
<td>8558</td>
<td>192340843</td>
<td>0.011%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16384</td>
<td>114211</td>
<td>2708480960</td>
<td>used for model</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20480</td>
<td>142360</td>
<td>3521951161</td>
<td>0.077%</td>
</tr>
<tr>
<td>EM3D</td>
<td>Electromagnetic wave propagation in a 3D object</td>
<td>100 50 75</td>
<td>9006</td>
<td>1099346</td>
<td>used for model</td>
</tr>
<tr>
<td>(Olden)</td>
<td></td>
<td>200 50 75</td>
<td>17950</td>
<td>1958089</td>
<td>0.06%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>400 50 75</td>
<td>35840</td>
<td>3751490</td>
<td>used for model</td>
</tr>
<tr>
<td></td>
<td></td>
<td>600 50 75</td>
<td>53700</td>
<td>5605689</td>
<td>0.58%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>800 50 75</td>
<td>71600</td>
<td>7423640</td>
<td>0.036%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000 50 75</td>
<td>89508</td>
<td>9246714</td>
<td>0.44%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1200 50 75</td>
<td>107250</td>
<td>11068172</td>
<td>1.0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1750 50 75</td>
<td>156644</td>
<td>16081209</td>
<td>0.13%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2500 50 75</td>
<td>223788</td>
<td>22919623</td>
<td>0.56%</td>
</tr>
<tr>
<td>Health</td>
<td>Columbian health care simulation</td>
<td>5 500</td>
<td>174334</td>
<td>82163549</td>
<td>used for model</td>
</tr>
<tr>
<td>(Olden)</td>
<td></td>
<td>5 1000</td>
<td>347759</td>
<td>269611573</td>
<td>4.69%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 1500</td>
<td>521785</td>
<td>566020698</td>
<td>1.76%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 2000</td>
<td>695524</td>
<td>969229080</td>
<td>used for model</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 2500</td>
<td>869397</td>
<td>1478554179</td>
<td>1.07%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 3000</td>
<td>1043277</td>
<td>2095657341</td>
<td>1.69%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 3500</td>
<td>1217079</td>
<td>2819993574</td>
<td>2.28%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 4000</td>
<td>1389947</td>
<td>3651266437</td>
<td>1.77%</td>
</tr>
</tbody>
</table>

Another focus of our work is to test how accurate the predictions are across various cache sizes. It is simple to understand that large caches have higher hit rates. However, the larger a cache is the slower its access time, and since the higher levels of the cache are frequently accessed by the processor, a low access time is desired. Figure 6 shows the comparisons of cache miss rate predictions to actual miss rates when the size of the cache varies. For each benchmark, the data size we use for this comparison is the one that has the largest error of those in Figure 4. Figure 7 shows the average relative hit rate error for each benchmark as associativity varies for each comparison in Figure 6.

These graphs show that even when the estimate is slightly off as data sizes vary, the estimations as cache sizes vary matches the pattern of the actual miss rates. Figure 7 shows that for all of the benchmarks the relative error for the hit rate has increased, but this is due to the same problem with the relative error involving HEALTH in Figure 5 where a small hit rate results in a much larger relative error. Another interesting result from this comparison is that the models achieve a more accurate estimate for 8 way associative caches than fully associative caches. The reason the fully associative cache appears to deviate from the estimate more than the set associative is that the fully associative cache has more values included in the calculations for the error when the cache size is small, increasing the average miss rate, subsequently increasing the relative error.

5 Possible Applications

The ability to accurately estimate miss rates over a large range of data and cache sizes has applications in several areas including but not limited to system configuration, benchmark set design, and dynamic cache control.

---

2 An 8-way associative cache must have at least 8 cache blocks, but a fully associative cache can be as small as one cache block, making it equivalent to a direct mapped cache.
Table 2: Benchmarks Available Online

<table>
<thead>
<tr>
<th>Suite</th>
<th>Benchmark</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>EuroBen</td>
<td>x.mod2a</td>
<td>N/A</td>
</tr>
<tr>
<td>SPEC</td>
<td>Applu</td>
<td>0.31%</td>
</tr>
<tr>
<td></td>
<td>Apsi</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Gcc95</td>
<td>0.14%</td>
</tr>
<tr>
<td></td>
<td>Swim</td>
<td>0.20%</td>
</tr>
<tr>
<td></td>
<td>Tomcatv</td>
<td>0.30%</td>
</tr>
<tr>
<td>Olden</td>
<td>BH</td>
<td>0.52%</td>
</tr>
<tr>
<td></td>
<td>EM3D</td>
<td>0.92%</td>
</tr>
<tr>
<td></td>
<td>Health</td>
<td>4.91%</td>
</tr>
<tr>
<td>NAS</td>
<td>BT</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>IS</td>
<td>N/A</td>
</tr>
<tr>
<td>NPB</td>
<td>LU</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>SP</td>
<td>0.47%</td>
</tr>
<tr>
<td>Other</td>
<td>ADI</td>
<td>0.50%</td>
</tr>
<tr>
<td></td>
<td>FFT</td>
<td>0.24%</td>
</tr>
</tbody>
</table>

Accuracy is the average relative hit rate error across all cache sizes for the Olden benchmarks, and the average relative hit rate error for a 64K cache for all other benchmarks.

5.1 Memory Hierarchy Design and Price/Performance Balancing

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. Given the range of data sizes of an application, customers can quickly determine a balance between price and performance when deciding on their memory configurations. The information would also provide the same insight to existing systems that are being used to execute new or larger applications. The customers can easily generate a model of the new applications for the system, and see for themselves how the miss rate changes with larger data sizes, and whether the current configuration would provide reasonable results. If the system needed an upgrade, the utility could show whether the new investment could actually decrease the miss rate or simply waste money. Last but not least, the tool can evaluate the cache performance of data sets too large to run on any existing machine. It may show whether the development of larger machines will allow execution of certain large data sets, or whether memory constraints will continue to prevent realistic execution times.

5.2 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. This saves time because the execution, although shorter, still yields the same miss rate as longer executions using larger data inputs. When evaluating a program transformation, the tool can construct a model for a program before and after the transformation. The evaluation will be thorough because the effect is shown on all program inputs not just a few data or cache sizes.

5.3 Dynamic Cache Control

Future memory hierarchy is likely to be adaptive. An example is a cache design by Balasubramonian et al. that allows alterations in cache size during execution [1]. Rather than searching numerous cache
configurations for one with the best IPC, one can statically determine the cache size that provides the best possible combination of performance and power savings. The system could also be sensitive to a dynamic environment. For example a low battery or a higher CPU temperature would lead to the selection of a smaller L1 cache in an attempt to reduce power consumption, and thus generate less heat. However, a draw back of this method is that the current model selects the cache size for the entire execution rather than adapting cache size for different program phases. We may instead predict miss rates on a per method basis, rather than for the program as a whole, following the observation of Huang et al. that program phases will often be defined by subroutine calls [9]. Alternatively, we can divide an execution into long instruction sequences that may span loops or multiple function calls using the methods of Magklis et al. [10] or Hsu and Kremer [6]. Whichever method is used, care must be taken to balance the overhead of prediction with its precision and to address the issue of compulsory misses since a large portion of the data needed for a program phase may not already be in the cache. By choosing the appropriate cache configuration for each phase, we may achieve comparable power savings and/or performance improvement without time and energy consuming search during execution.

6 Related Work

A large number of tools have been developed to analyze cache or memory performance among other metrics. Our tool focuses exclusively on the memory performance of a benchmark, discarding how the miss rate affects the benchmark’s performance. This may provide a misleading view of program performance. It is obvious that a program with a high miss rate may perform well if the latency is well hidden by other useful work. However, by focusing only on cache behavior, our tool provides highly accurate prediction across cache sizes and program input sizes.

Few tools provide reliable prediction for a program across all data inputs. Crovella and LeBlanc showed how to predict future executions with respect to cycles wasted waiting on other processors [2]. Other tools such as HPCView [7] by Mellor-Crummey et al. and SvPablo and HPM by DeRose et al. [8, 3] provide insight for future executions but not accurate prediction of future cache miss rates, although they may, after many executions, generate estimates across all practical inputs.

Our tool uses Java applets and can be run on any machine that supports Java JVM and Java 3D library.
Figure 6: Miss rates comparison for BH, EM3D, HEALTH with constant data size
HPCView can also be accessed from the web, but requires a much larger space. In comparison a single model used by our tool for a benchmark occupies less than 15kB, but information for a single application using HPCView has resulted in 10,000 files occupying 300MB [7]. One disadvantage of being on the web is that dynamic verification of estimates is not possible without users accessing the source and recompiling the benchmark for their environment, since the benchmarks are written in languages other than Java. The absolute miss rates are also compiler and machine dependent. However, the topology of the miss rate prediction should be consistent across different machines and also compilers in the absence of large scale program reorganizations.

By limiting our work to whole program miss rates, we have removed the complexity associated with correlating binaries to original source code [14]. Since our models are generated using instrumented binaries, the complexities of compiler optimizations are fully considered by our prediction. However, this does present a disadvantage when compared to other tools such as HPCView’s and Carnival’s graphical interface [7, 15] that correlate frequent cache misses with specific program instructions, allowing a user to identify the cause of a performance degradation for that run. However, this approach is orthogonal to the basis of our work. Our tool can be extended to predict cache behavior for smaller program units such as functions or data structures and consequently be embedded into program-based tools.

A large number of tools such as SvPablo [8], HPM [3] and Carnival [15] evaluate memory performance for both sequential and parallel applications. Our tool is currently used exclusively to estimate memory performance on sequential programs. The high accuracy suggests that many programs have a predictable cache behavior. However, it is yet an open question whether the memory performance of parallel programs is amenable to a similar type of analysis, where the number of processing units will be a third dimension in addition to cache and input data size.

7 Summary

This paper presents a tool for estimating and graphically displaying the miss rate of a program for all its input and cache sizes. After only two executions, our model can estimate the miss rate of a fully associative cache of any size for any input. The application allows the user to also reverse the process by finding the input or cache size needed to generate a miss rate given a cache or input size respectively. Prior work showed that our model was accurate to 99% for fully associative caches, and better than 98% accurate for caches with limited associativity when compulsory cache misses were excluded. This work demonstrated that this
high level of accuracy is maintained for benchmarks with dynamic data structures, even when the error is compounded by estimating across various cache sizes.

8 Acknowledgments

This work is supported by the National Science Foundation (Contract No. CCR-0238176, CCR-0219848, and EIA-0080124) and the Department of Energy (Contract No. DE-FG02-02ER25525).

References


