Statstack: Efficient Modeling of LRU caches

David Eklov and Erik Hagersten
Department of Information Technology
University of Uppsala
{david.eklov,eh}@it.uu.com

Abstract

The identification of the memory gap in terms of the relatively slow memory accesses put a focus on cache performance in the 90s. The introduction of the moderately clocked multicores has shifted this focus from memory latency to memory bandwidth for modern processors. The multicore's limited cache capacity per thread in combination with their current a projected off-chip memory bandwidth limitation makes this the most likely bottleneck of future computer systems.

This paper presents a new and efficient way of estimating the cache performance for an application. The method has several similarities with that of Stack Distance, but instead of counting *unique memory objects*, as is done for Stack Distance calculations, our schema only requires the *number of memory accesses* to be counted between two successive accesses to the same data object. This task can be efficiently handled at runtime by existing built-in hardware counters. Furthermore, only a small fraction of the memory accesses have to be monitored for an accurate estimation.

We show how low-overhead runtime data, similar to that of StatCache, is sufficient to feed this model. We evaluate the accuracy of the proposed transformation based on sparse data and compare the results with that of native stack distance based all memory accesses. We show excellent accuracy over a wide range of cache sizes and applications.

1 Introduction

In pace with the widening memory gap of the past, cache behavior and application locality has earned plentiful of attention. Since the introduction of multicores, the slower CPU frequency improvement has reduced the pace of the widening of this gap, but has instead put the focus on memory bandwidth as one potential performance bottleneck for future architectures.

Both memory latency and memory bandwidth issues calls for powerful techniques leading to insight into application cache behavior. Still, these techniques have to be efficient enough to evaluate long-running applications operating on huge input data sets.

One such powerful technique is stack distance, introduced by Matssson [15]. The stack distance of a memory operation is the unique number of memory object touched since the memory object touched by the operation was last accessed. Often, the memory object size chosen is the cache line size of a cache. Based on the stack distance distribution from an application, the miss rate of a fully associative cache can be calculated trivially and quickly. Since the introduction of stack distance, this abstraction has been used in plenty of research work [8][12][21].

While the stack distance technique is powerful, it does not easily lend itself for efficient implementation. The requirement to count the number of unique memory objects accesses for every memory re-use of an application results very state-full implementations. This requires elaborate book-keeping of all the memory objects accessed. Furthermore, in order to warm-up the cache, this bookkeeping must be maintained for many millions of memory references before long stack distances can accurately me measured.

The research presented in this paper is inspired by StatCache [3][4]. StatCache is a statistical cache modeling techniques based on to runtime information which can be collected with a low runtime overhead. Instead of keeping track of the number of *unique memory objects* touched in between any two consecutive access to a memory object, StatCache simply records the *number of memory accesses* between the two consecutive accesses. In addition, Statcache does not require a large number of consecutive accesses to be monitored. Furthermore, StatCache only collects this information for a very sparse selection of memory accesses. For example, one access in every 100.000th accesses can randomly be selected to be monitored.

This runtime task can be performed at almost no overhead by the readily available hardware counters and watchpoint mechanisms provided by modern CPUs and operatring systems. Practical experiments using this method report an average runtime overhead of 60 percent [4]. Off-line mathematical methods are used to calculate the miss rate for any sized *fully associative random-replacement* caches using mathematical solver

in seconds.

A task of this research is to find a method to model a LRU caches, similar to the use of stack distance data, but to base that modeling on low-overhead runtime data similar to that of Statcache. At a first glance it appears that the sparsely collected data is a poor fit for building up the necessary access state needed to model LRU. A completely new approach was needed.

Instead of counting the exact number of unique memory objects between two consecutive accesses to the same piece of data, we capture information about the number of intervening memory accesses and the distribution of how soon again the data they touch will be touched again.

The paper continues by looking at Statcache and its efficient data collection in review. Next, we give a high-level description of the new LRU modeling followed by a rigorous description of its algorithm. We also discuss possible sources of errors introduced in the model and describe a scheme for collecting runtime data that will minimize these. We evaluate the accuracy of the new model by comparing its results with the output from a traditional cache simulator fed with full address traces for 25 applications. We show that a sparse sample consisting of only 500 000 samples from each application can be used to accurately model a fully-associative LRU cache across a wide range of cache sizes. A sensitivity analysis is performed to isolate some of the error sources discussed earlier. Lastly we study the accuracy of the new model if only 100 000 samples were used.

2 Statcache in review

Statcache is a tool for estimating an application's cache miss ratio [3][4]. It uses an online sampler that attaches to the target application and measures the reuse distances of a set of randomly selected memory references. The reuse distance of a memory reference A is equal to the number of memory references in the target application's address trace between A and the previous memory reference to the same cache line. The sampler records the set of measured reuse distances and some meta data about the sampled memory references in a structure called a reuse distance sample, RDS. The RDS is feed as input to an offline statistical cache model. The Statcache cache model models a fully associative cache with a random replacement policy. Accurately estimated miss ratios are demonstrated using sparse RDS:s collected with sample rates as low as 10^{-6} [4], i.e., only containing reuse information of every one millionth memory access.

Note that the reuse distance of a memory access A, as defined by Berg [3], is not the same as it stack distance, as defined by Mattson [15], which is the number of distinct cache lines accessed by the application between A and the previous memory access to the same cache line.

Contrary to online collection of stack distances, the collection of reuse distances allows the sampler to make efficient use of functionality supported by contemporary hardware and operating systems, such as hardware performance counters and watchpoints. The Statcache sampler allows for sampling of unmodified binaries and has a small impact on the sampled applications runtime. Berg et al. [4] reports an average slowdown of only 60 percent when hardware and operating system support are used.

The Statcache sampler works as follows: Hardware counter overflow traps are used to halt the execution when a memory references that has been selected for sampling executes. A one-time watchpoint is set for the address it accesses. The number of memory accesses executed to date is recorded by reading a hardware counter, after which the computation is continued. The next time the same address is accessed, a watchpoint trap is generated, which again halts the execution. The reuse distance is calculated as the difference between the number of memory accesses executed to date and the previously recorded number.

3 Statstack

Statstack, like Statcache, is a tool for estimating an applications cache miss ratio. Statstack introduces a new statistical cache model that, unlike the statistical cache model of Statcache, models a fully associative cache with an LRU replacement policy. The Statstack cache model uses the same kind of input data, a reuse distance sample, RDS, as the cache model of Statcache. This allows Statstack to use the same efficient sampling technique as Statcache, and thereby inherits Statcache's low runtime overhead.

Modeling LRU caches based on the sparse information in an RDS may appear to be an impossible task at first. A traditional cache simulation approach requires knowledge about every memory reference in order to maintain the LRU ordering used to model the replacement strategy. Mattson's stack-distance approach needs to keep count of the number of unique memory references, which also requires all memory reference to be monitored.

Our approach has instead been tailored to fit the sparse information in an RDS: the reuse distance for a limited, but representative, selection of memory references, i.e., how many memory references are performed between two successive references to a specific cache line. Based on this information we perform two steps: 1) we put together the typical reuse distance distribution for the memory references of the application; 2) for each reuse distance in the RDS we use that distribution to determine the likelihood that each of the memory references performed between the two successive references

to the cache line is accessed more than once. Based on this likelihood, we can estimate the number of unique cache lines touched during the reuse distance, i.e., its stack distance.

In the remainder of this section, we will derive the cache model of Statstack. In Section 3.1, we introduce a relation between reuse distance and stack distances that allow us to compute the stack distance of a memory references given that we know its reuse distance and the reuse distances of the memory references executed before it. In Section 3.2 we define the expected stack distance of a memory reference and derive an analytical expression for it in terms of reuse distances. In Section 3.3 we discuss an approximation that allow us to estimate expected stack distances based on the reuse information in a sparse RDS. Next, in Section 3.4, we show how to compute the cold miss ratio from the reuse information in a sparse RDS. Finally, in Section 3.5, we put the pieces together and show how to estimate an application's capacity miss ratio based on expected stack distances computed form the reuse information in a sparse RDS.

3.1 Stack Reuse Relation

In this section, we introduce a fundamental relation between reuse distances and stack distances. This relation will allow us to compute an application's stack distances from its reuse distances. Before proceeding, we need to make our terminology more precise.

Definition 3.1 *Let A and B be two consecutive memory access to the same cache-line-sized piece of memory in an arbitrary address trace where A is executed prior to B, then the*

- ***reuse distance*** *of B is the number of memory accesses executed between A and B,*
- ***forward reuse distance*** *of A is the number of memory access executed between A and B,*
- ***stack distance*** *of B is the number of distinct cache lines accessed by memory accesses executed between A and B.*

The relation between reuse distance and stack distance is best explained by an example.

Example 1: Figure 1 shows a sequence of an address trace, where memory references are marked by circles on the horizontal time axis. The memory references are labeled according to their position in the address trace. The memory reference x_8 to cache line l_1 has a stack distance of three, since there are three distinct cache lines l_2, l_3 and l_4 accessed by the memory references $x_2, ..., x_7$. The only memory reference $x_2, ..., x_7$ with forward reuse distances greater than their distance to x_8, measured in intermediate memory reference, are x_4, x_5 and x_7 (their arcs "reach" beyond x_8). We note that, the stack distance of x_8 is equal to the number of the memory references $x_2, ..., x_7$ whose forward reuse distance is greater than their distance to x_8. We further note that, x_4, x_5 and x_7 are the last of the memory references $x_2, ..., x_7$ to accesses the cache lines l_2, l_3 and l_4.

As we will see, it is not hard to realize that the result of the above example holds true in general. We introduce the following notation, let $R(x_t)$ and $\vec{R}(x_t)$ denote the reuse distance and the forward reuse distance of x_t respectively. Further, let $Q(x_t)$ be the set of memory references between x_t and the previous memory reference to the same cache line. We can now state the result of Example 1 in a more general form as follows: The stack distance of a memory reference x_t is equal to the number of memory references, $x_i \in Q(x_t)$, for which $\vec{R}(x_i) + i > t$. This statement follows from the observation that $\vec{R}(x_i) + i > t$ if and only if x_i is the last of the memory references in $Q(x_t)$ to access cache line l_i. The stack distance of x_t, denoted by $S(x_t)$ can now be expressed as,

$$
\begin{aligned}
S(x_t) &= \sum_{i=t-r}^{t-1} \mathbf{1}(\vec{R}(x_i) > t - i) \\
&= \sum_{j=1}^{r} \mathbf{1}(\vec{R}(x_{t-j}) > j)
\end{aligned} \tag{1}
$$

where, $r = R(x_t)$ and $\mathbf{1}(\alpha)$ is defined to be one if α is true and zero otherwise. The second equality in (1) is due to the variable substitution $j = t - i$. (1) can also be derived from equations presented by Bennett and Kruskal [2], for an ouline of a derivation see Appendix A.

3.2 Expected Stack Distance

To compute the stackdistance of a memory reference x_t using the method of Section 3.1 we need to know the forward reuse distance of all memory references executed between x_t and the previous memory reference to the same cache line. However, our goal is to approximate an application's miss ratio given only the sparse data in its RDS. To this end we introduce the concept of an expected stack distance.

In this section we, derive an analytical expression for the expected stack distance. This expression, as we will show in Section 3.3, lends itself for an approximation that allows us compute expected stack distances using only the sparse information in an RDS.

We define expected stack distance of a memory reference with a reuse distance of r to be the average stack distance of all memory references in an application's address trace with a reuse distance of r. A consequence of

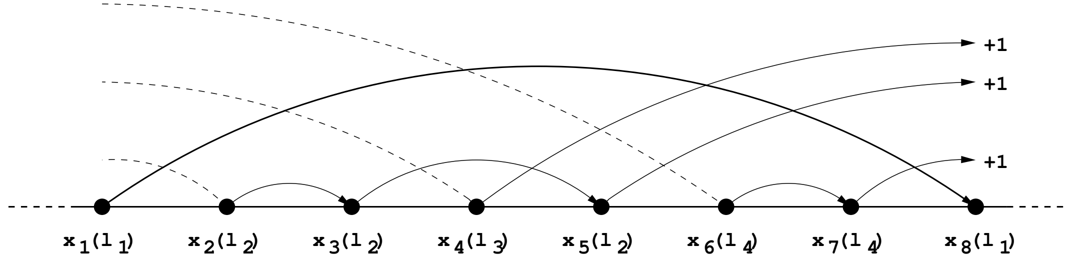

Figure 1: The figure shows a sequence of an address traces. Memory references are marked by circles on the horizontal time lines. The labels below the circles show the name of the memory references and the cache line accessed in parenthesis. The arcs connect consecutive memory references to the same cache line.

the above definition is that all memory references with the same reuse distance have the same expected stack distance. We denote the expected stack distance of the memory references with a reuse distance of r by $ES(r)$.

To derive an analytical expression for the expected stack distance we introduce the following notation. Let $T(r)$ be the set of all memory references with a reuse distance of r in the target application's address trace and let $n_r = |T(r)|$. We can now express the expected stack distance, $ES(r)$ as follows,

$$\begin{aligned} ES(r) &= \frac{1}{n_r} \sum_{x_i \in T(r)} S(x_i) \\ &= \frac{1}{n_r} \sum_{x_i \in T(r)} \sum_{j=1}^{r} 1(\vec{R}(x_{i-j}) > j) \\ &= \frac{1}{n_r} \sum_{j=1}^{r} \sum_{x_i \in T(r)} 1(\vec{R}(x_{i-j}) > j). \end{aligned} \tag{2}$$

To get the second equality in (2) we substitute $S(x_i)$ with the right hand side of (1), for the third equality we simply interchange the order of summation. The inner sum on the rightmost side of (2) is equal to the number of memory references x_{i-j} with a forward reuse distances greater than j. We denote this sum by $n_r(j)$ and can now write $ES(r)$ as follows,

$$ES(r) = \sum_{j=1}^{r} \frac{n_r(j)}{n_r}. \tag{3}$$

The following example shows how to compute the expected stack distance, first using the definition and secondly usign (3).

Example 2: Figure 2 shows an address trace in a similar fashion to Figure 1, but zooms in on the three memory references x_{i_1}, x_{i_2} and x_{i_3}. These memory references are assumed to be the only references in the address trace with a reuse distance of three. Using the method of Section 3.1 we can compute their stack distances, which are one, two and three respectively. We can now compute the expected stack distance of the three memory references using the definition of expected stack distance simply by averaging their stack distances, which gives us an expected stack distance of two ($\frac{1}{3}(1 + 2 + 3)$). To compute the expected stack distance of the three memory references using (3), we first need to compute $n_3(1)$, $n_3(2)$ and $n_3(3)$, which are three, one and two respectively. For example, $n_3(2)$ is one, since only one of the memory references x_{i_1-2}, x_{i_2-2} and x_{i_3-2} has a forward reuse distance greater than two, namely x_{i_1-2}. Carrying out the computation in (3) we again get an expected stack distance of two ($\frac{1}{3}(n_3(1) + n_3(2) + n_3(3)) = \frac{1}{3}(3 + 1 + 2) = 2$).

3.3 Approximation

To compute an expected stack distances using (3) requires more information than generally available in a sparse RDS. However, the structure of (3) lends itself for an approximation to better suit the nature of a sparse RDS. The idea is to approximate $\frac{n_r(j)}{n_r}$ terms in (3) based on the reuse information in a sparse RDS.

In this paper we are targeting a sample rate of between 10^{-4} and 10^{-5}, which means that the RDS:s only contain the reuse distances of on average one out of every $10,000$ to $100,000$ memory reference. To compute an expected stack distance with (3), we need to know the reuse behavior of very specific sets of memory references. A sparse RDS is therefore unlikely to contain enough information to allow us to accurately estimate the reuse behavior of these specific memory references. In order to make use of a sparse RDS, an approximation is needed.

Assume that we know the reuse distance of all memory references in the target application's address trace. We consider the following approximation,

$$\frac{n_r(j)}{n_r} \approx \frac{n(j)}{n}. \tag{4}$$

Here, $n(j)$ is the number of memory references with a reuse distance greater than j, and n is the total number of memory references in the address trace. The above

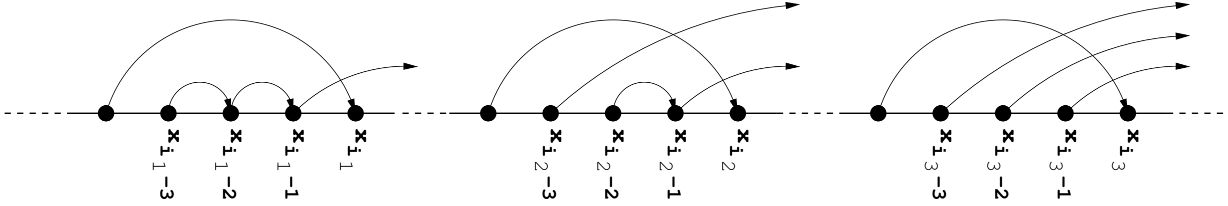

Figure 2: The figure shows a sequence of an address trace. Memory references are marked by circles on the horizontal time axis. The figure zooms in on the three memory references, x_{i_1}, x_{i_2} and x_{i_3} that have a reuse distance of three.

approximation allows us to compute expected stack distances without the constraint of having to know the reuse behavior of specific memory references. We discuss the implications of the above approximation in Section 5.

The above approximation allows us to estimate the $\frac{n_r(j)}{n_r}$ terms in (3) based on the reuse information in a sparse RDS as follows,

$$\frac{n(j)}{n} \approx \frac{\hat{n}(j)}{\hat{n}}. \tag{5}$$

Here, $\hat{n}(j)$ is the number of memory references with a reuse distance greater than j, and $\hat{n}$ is the number of reuse distances in an RDS. In Section 5 we discuss the implications of the above approximations, and in Section 6 we evaluate their accuracy. We can now write the following expression for the expected stack distance,

$$ES(r) \approx \sum_{j=1}^{r} \frac{\hat{n}(j)}{\hat{n}}. \tag{6}$$

3.4 Cold Misses

When a memory reference that the Statcache sampler has selected for monitoring executes the sampler sets a one-time watchpoint for the address being accessed. If this address is not accessed again, the watchpoint will never trigger. When the application has finished its execution, the untriggered watchpoints are recorded in the RDS as dangling samples.

The number of cold misses experienced by an applications is proportional to the number of dangling samples [5]. It can be explained as follows. A cold miss occurs when the application accesses cache-line-sized piece of memory for the first time. Every cache-line-sized piece of memory accessed by the application is also accessed one last time and if the last memory references is sampled it will result in a dangling sample. Since the sampler samples all memory accesses with equal probability, the cold miss ratio is equal to the number of dangling samples divided by the number of reuse distances in the RDS.

3.5 Cache Model

In this section, we put the pieces together and show how the Statstack cache model estimates an application's miss ratio including both capacity and cold misses from its input data, an RDS collected by the Statcache sampler.

Armed with equation (6), we can compute the expected stack distance distribution of an application given its RDS as follows: First, we compute the expected stack distance of each distinct reuse distance in the RDS using (6). Then, by weighting each of the expected stack distances with the frequency of the corresponding reuse distance in the RDS we get the expected stack distance distribution.

To compute the application's capacity miss ratio for a given cache size C, we simply compute the fraction of stack distances in the expected stack distance distribution that are greater than C. Here, C is measured in cache lines. Finally, by adding cold miss ratio, computed as shown in Section 3.4, to the capacity miss ratio we get the application's miss ratio for a fully associative cache with an LRU replacement policy.

By computing the miss ratio using expected stack distances the cache model gets susceptible to program phase changes [7][18]. However, the Statcache sampler implements an hierarchical sampling policy, discussed in Section 4, that divides the target application's address trace into time windows. This allows us to estimate the miss ratio for each time window individually, which effectively reduces the cache models sensitivity to program phase changes.

Figure 3 shows miss ratios estimated by Statstack as a function of cache size for the SPEC CPU2006 benchmarks, together with a reference miss ratio obtained from a trace driven cache simulator. The RDS:s used to estimate these miss ratios have been collected using a sample rate of 10^{-4}, i.e. the RDS:s contains the reuse distances of only one out of every $10,000$ memory references. The experimental setup used to obtain these results is described in detail in the Section 6.

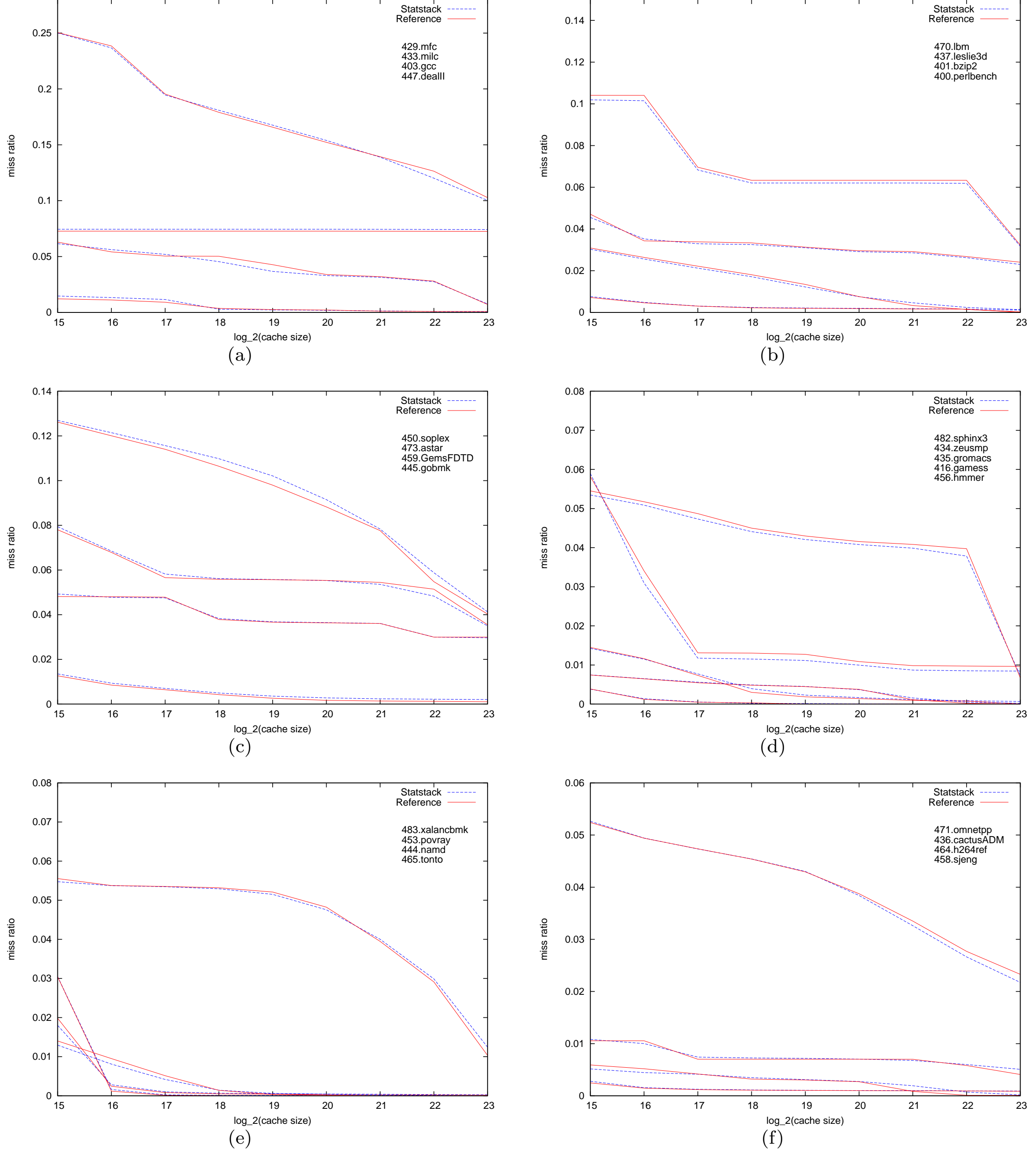

Figure 3: The graph show the miss ratio estimated by Statstack as a function of cache size for the SPEC CPU2006 benchmarks (labeled Statstack), together with a reference miss ratio obtained from a trace driven cache simulator (labeled Reference). The RDS:s used by Statstack was collected with a sample rate of 10^{-4}.

4 Hierarchical sampling

In this section we review a hierarchical sampling policy implemented by the Statcache [4]. As we will see in Section 5, Statstack leverages the hierarchical sampling policy to reduce the cache models sensitivity to phase changes. The hierarchical sampling policy also reduces the execution overhead inflicted by the Statcache sampler on the target application.

The hierarchical sampling policy works as follows. The sampler divides the target application's address trace into time windows. It uniformly samples the memory references executed in each window, using the techniques discussed in Section 3, and stores their reuse dis-

tance in separate RDS:s, one for each time window. By feeding these RDS:s one by one to the Statstack cache model, we can estimate the miss ratio for each of the time windows individually.

To reduce the execution overhead inflicted on the target application, the sampler inserts random sized time gaps between the time windows. This allows the target application to run at close to native speed during the time gaps between windows.

5 Error Sources

In Section 3 we used three approximations to derive the cache model. These approximations are all potential sources of errors. A forth approximation is introduced by the hierarchical sampling policy. We have grouped these error sources into two categories, approximation errors and statistical sampling errors. These categories are further divided into, type I and type II approximation error and type I and type II statistical sampling errors. In the remainder of this section we discuss these error sources one by one.

5.1 Type I Approximation Error

Statstack computes and uses the expected stack distance distribution of the target application to approximate its miss ratio. By doing this, Statstack essentially approximates the application's stack distance distribution with its expected stack distance distribution. Since these two distributions are generally not the same, this approximation can be a potential source of errors, which we call the type I approximation error.

Figure 4 shows an example stack distance distribution of memory references with a reuse distances of r and also their expected stack distance. The memory references with a stack distance greater than the cache size will result in a cache miss. As shown in Figure 4 the expected stack distance of the memory references is less than the cache size. The miss ratio calculated from the expected stack distance distribution is therefore zero. However, the miss ratio calculated from the stack distance distribution is not zero.

The above error grows large if the application makes a large number of memory references with the same reuse distances but with largely varying stack distances. This is most likely to be the case for applications having program phases with largely different cache behaviors. However, as we will see in Section 6, this type of error can be made fairly small by dividing the application's address trace into time windows, using the hierarchical sampling policy of the Statcache sampler.

5.2 Type II Approximation Error

We call the impact of (4) on the cache models accuracy the type II approximation error. When applying (4) we lose all information of the order in which the sampled memory references appear in the target application's address trace. If the reuse distances of the target application's memory references display certain patterns, the cache model can perform poorly. For example, the short reuse distances may all occur in the beginning of a time window rather than to be evenly distributed throughout the time window, as we assume with our model. However, as we will see in Section 6, the miss ratios estimated for the studied applications does not seem to suffer from large errors due to this approximation.

5.3 Type I Statistical Sampling Error

We call the impact of (5) on the cache models accuracy the type I statistical sampling error. Here, we are trying to approximate the frequency of memory references with a reuse distances greater than j in the target application's address trace, with the frequency of reuse distances greater than j in a sparse RDS. This gives rise to the type I statistical sampling error.

The magnitude of the type I statistical sampling error depends on the sample rate used by the Statcache sampler, i.e. the number of reuse distance in the RDS, and the variance of the target application's reuse distance.

5.4 Type II Statistical Sampling Error

The hierarchical sampling policy introduces a second type of statistical sampling error, which we call the type II statistical sampling error. The hierarchical sampling policy inserts time gaps between the time windows to reduce to overhead of the Statcache sampler. This makes the hierarchical sampling policy similar to time sampling [14][13][20]. The idea of time sampling divide the target application's address trace into time windows, but instead of measuring reuse distances of randomly selected memory references, complete address traces are collected for each window. These traces are then feed one by one to a trace driven cache simulator. The type II statistical sampling errors can be analyzed using similar techniques as used for time sampling [20].

6 Evaluation

This section evaluates the accuracy of the Statstack cache model by comparing the miss ratio estimated by Statstack to references miss ratio obtained using a traditional trace driven cache simulator. We also discuss an empirical method used to find the parameters of the hierarchical sampler that minimize the the approximation and sampling errors.

6.1 Experimental Setup

We use 28 out of the 29 programs in the SPEC CPU2006 benchmark suite (481.wrf did not compile on our system) run with their first reference input sets. All benchmark programs are compiled using GCC version 4.1.2

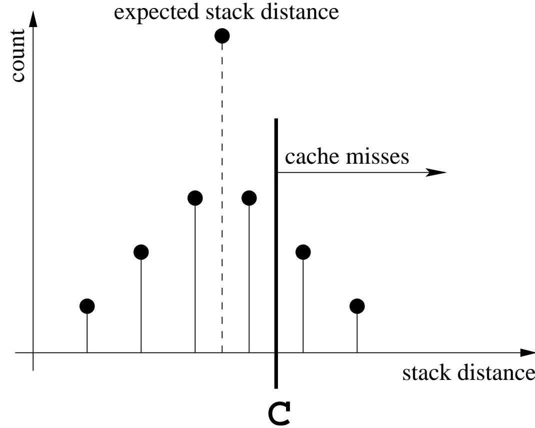

Figure 4: The figure shows the distribution of stack distances of a set of memory references with a reuse distance of r in the form of a histogram. The vertical line labeled C represents the cache size.

with optimization level 02 targeting an x86_64 system.

In order to evaluate the accuracy of our model, we first collect address traces from the benchmarks programs using an in-house instrumentation tool. The address trace collection is started after approximately 60 seconds of uninstrumented execution for all programs except for 403.gcc whose total execution time with its first reference input set is less than 60 seconds. All traces used for evaluation contains 5×10^9 memory access. This trace size was chosen for practical reasons.

To obtain reference miss ratios we use a trace-driven cache simulator, configured to simulate a fully associative cache with an LRU replacement policy for cache sizes ranging from 32kB up to 8MB. The reference miss ratios include both cold and capacity misses.

We then collect sparse RDS:s by measuring the reuse distances of randomly selected memory references in the address traces using the hierarchical sampling scheme. We finally use an implementation of the Statstack cache model, as described in Section 3, to estimate the miss ratios of the traces for cache sizes ranging from 32kB up to 8MB.

6.2 Sampler Parameters

The hierarchical sampling policy, discussed in Section 4, exposes three parameters to the user, window size (d), samples per window (s) and number of windows (w). The choice of these parameters, called the sampler parameters, influences both the accuracy of the cache model and the runtime overhead of the sampled application. In this section, we formulate an optimization problem, whose solution gives us an optimal set of sampler parameters, for a given runtime overhead.

Before proceeding, we need to define how to measure the error of the cache model. In this study, we use a mean square error, MSE, which is the mean of the squared differences between the estimated and the reference miss ratios for cache sizes ranging from 32kB to 8MB in increments of 4kB. Let $\hat{mr}(x)$ and $mr(x)$ be the estimated and the reference miss ratio, then the MSE can be expressed as follow,

$$MSE = \sum_{i=8}^{2^{11}} (\hat{mr}(2^{12}i) - mr(2^{12}i))^2.$$

The execution overhead inflicted by online sampler when the target application is executing within a time window is proportional to the number of reuse distances measured in the time window, s. Let N be the total number of reuse distance measured in all time windows. We have the following relation $N = s \cdot w$.

Given a runtime overhead, expressed in terms of N, $N = s \cdot w$ together with s, $w > 0$ and $0 < d < M$, where M is the number of memory references in the taget application's address trace, gives us a set of constraints on the sampler parameters. We note that d, the window size, does not influence the runtime overhead.

To define an objective function we assume that we have set of training applications, B, whose cache behaviors represents the cache behaviors of the applications for which Statstack is intended to be used. In this study, we used the SPEC CPU2006 benchmark as the training set. We can now setup an objective function,

$$\max_{b \in B}\{MSE(b)\}$$

By minimizing the above objective function we minimize the MSE of the application in B, with the largest MSE. Note that, the MSE is a function of the sampler parameters.

To find a solution to the above optimization problem we pick a set of uniform points in the search space, compute the objective function for all these points, and select the parameters that minimize the objective functions on the search space.

Using the above method we found the following sampler parameters for $N = 100,000$, we have $d = 10^6$, $s = 1500$, $w = 67$, and for $N = 500,000$, we have $d = 10^6$, $s = 1500$, $w = 333$.

6.3 Sensitivity Analysis

In this section, we present results of two sets of experiments. The first, Experiment I, shows how the estimated miss ratio is affected by the approximation errors. The second, Experiment II, shows how the estimated miss ratio is affected by the statistical sampling errors.

6.3.1 Experiment I

In this experiment, we evaluate the impact of the approximation errors on the accuracy of the cache model. In order to do this, we collect RDS:s that contain the reuse distance of all memory references in the address traces, and estimate miss ratios based on these RDS:s. Since the RDS:s contain the reuse distance of all memory references, we effectively eliminate the statistical sampling errors. By comparing these miss ratios to the reference miss ratios we can evaluate the impact of the approximation errors on the accuracy of the cache model in isolation.

The RDS:s that we use in this experiment are collected using a window size of $d = 10^6$, which was found to be the best window size in Section 6.2. Figure 5 shows the estimated miss ratio next to the reference miss ratio for all benchmark (except for 429.mfc[1], 433.milc[1], 471.omnetpp[1], 410.bwaves[2] and 453.povray[2]).

The close agreement between the estimated and reference miss ratios indicates that the approximation errors' impact on the cach model's accuracy is small. The largest discrepancy between the estimated and the reference miss ratios can be seen for small cache sizes of less than 64KB. This is likely due to the type I approximation error. For most applications the distribution of stack distance are heavily weighted towards short stack distances. This makes it likely that there are a large number of memory references with the same reuse distance, that display the behavior shown in Figure 4.

The application with the largest error for large cache sizes is 473.astar for which the estimated miss ratio is somewhat lower than the reference for 4MB caches. This indicates that some portion of the expected stack distances that are close to the 64k (4MB / 64) are under estimated.

[1]Due to the large dispersion of reuse distance, the amount of memory needed for simulation exceeds the amount of memory available on our system.

[2]Both the reference and the estimated miss ratio are zero for cache sizes larger than 32kB.

6.3.2 Experiment II

In this experiment, we evaluate the impact of the statistical sampling errors on the accuracy of the cache model. In order to do this, we collect RDS:s and estimate miss ratios for each address trace several times. The dispersion between the estimated miss ratios measures the statistical sampling errors.

For each benchmark we collect RDS:s and estimate miss ratios, 32 times with the sampler parameters found in Section 6.2, for both $N = 100,000$ and $N = 500,000$ (except for 471.omnetpp[1], 410.bwaves[2] and 453.povray[2]). Figure 6 shows three graphs for each of the benchmarks, Statstack-min, Statstack-max and Reference. Statstack-min and Statstack-max shows the minimum and the maximum miss ratio out of the 32 estimated miss ratios, for $N = 500,000$ and Reference shows the reference miss ratio. Figure 7 shows the same type of graphs, but for $N = 100,000$.

The distance between the Statstack-min and the Statstack-max, is indicative of the dispersion of the estimated miss ratios and therefore a measure of the impact of the statistical sampling errors. As shown in Figure 6 the distances between the Statstack-min and the Statstack-max graphs are relatively small for most of the benchmarks, when the size of the RDS:s are $N = 500,000$. We can therefore expect the Statstack cache model to estimate accurate miss ratios when RDS:s of size $N = 500,000$ are used. However, as Figure 7, the distance are larger for RDS:s of size $N = 100,000$. For the miss ratios shown in Figure 7, only 67 windows are sampled, as opposed to Figure 6 where 333 windows are sampled. The larger dispersion shown in Figure 7, is therefore due to the type II statistical sampling error.

6.4 Cache Model Performance

The overall performance of cache models, like Statstack, has two parts, the performance of the data collection mechanism and the performance of the model itself.

The Statstack cache model uses the Statcache sampler to collect its input data. The sparseness of the collected data in conjunction with the use of hardware and operating system support makes the execution time overhead of the target application small.

The execution time of the cache model, for the RDS:s used for evaluation, is only a few seconds. The short execution time is due to the sparseness of the input data. Internally, the cache model stores the reuse distances in a histogram. The most time consuming operation for the cache model is to sort the data in the input RDS into the histogram, but because of the sparseness of the RDS:s this operation is still fast. When the histogram is build, the cache model requires only two passes over the buckets to compute the miss ratio. The number of buckets in the histogram is equal to the number of

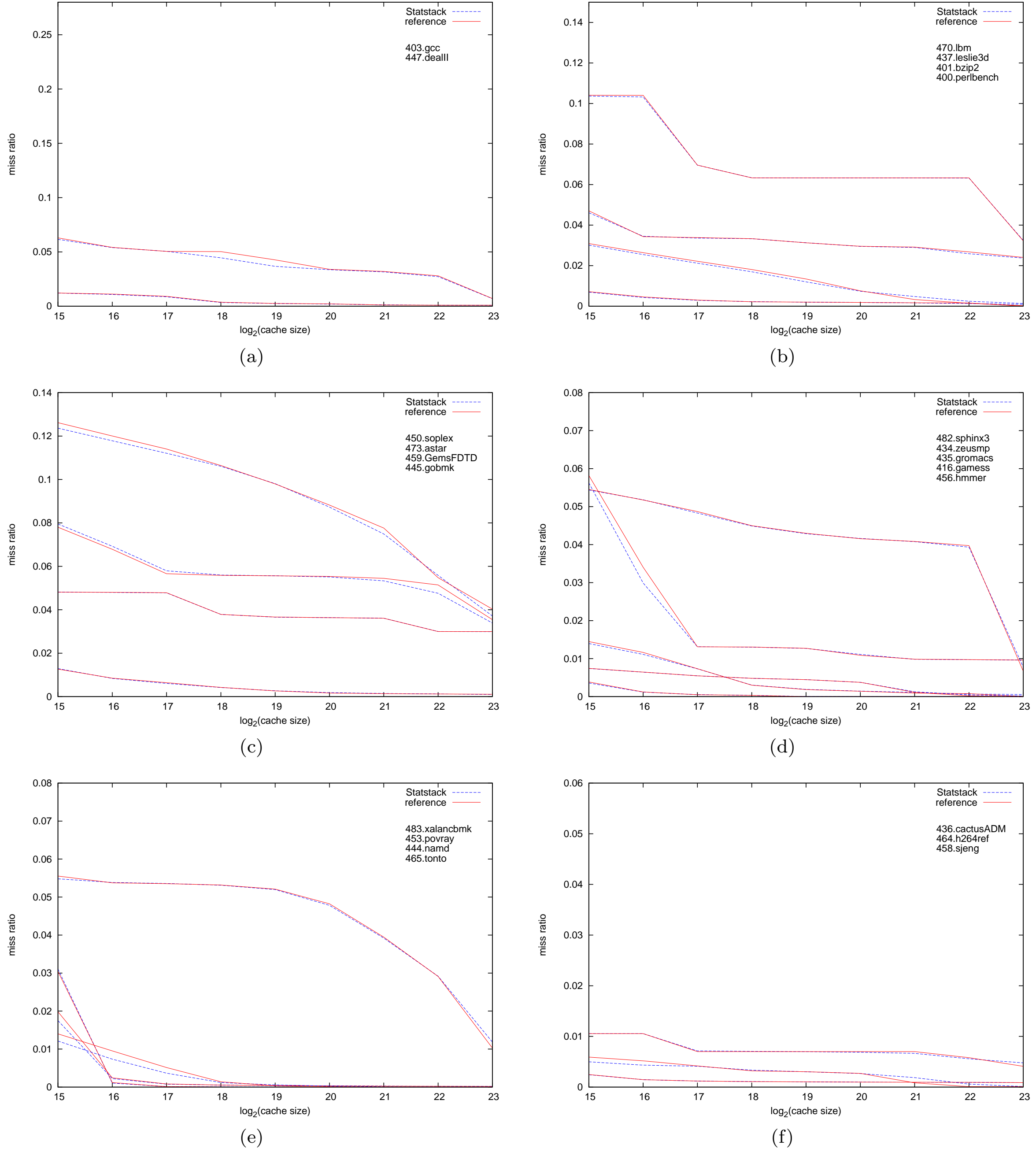

Figure 5: Miss ratio as a function of cache size. The graph labeled Statstack shows a miss ratio estimated using RDS:s containing the reuse distance of all memory accesses in the address trace. The graph labeled Reference shows the reference miss ratio. The graphs are ordered by their y intercept according to the list on the right.

distinct reuse distance in the RDS.

7 Related Work

Since the introduction of stack distance, by Mattson [15] in the early 70s, it has earned plenty of attention by researchers in quest to find efficient techniques to study cache locality. Initially, Mattson proposed a stack based algorithm to measure the stack distances of memory accesses in an address trace. To improve the efficiency of the stack based algorithm different data structures have been used to maintain the count of distinct data objects accessed between two consecutive accesses to the same data object [2][1]. Approximate algorithms where efficiency is traded for accuracy have

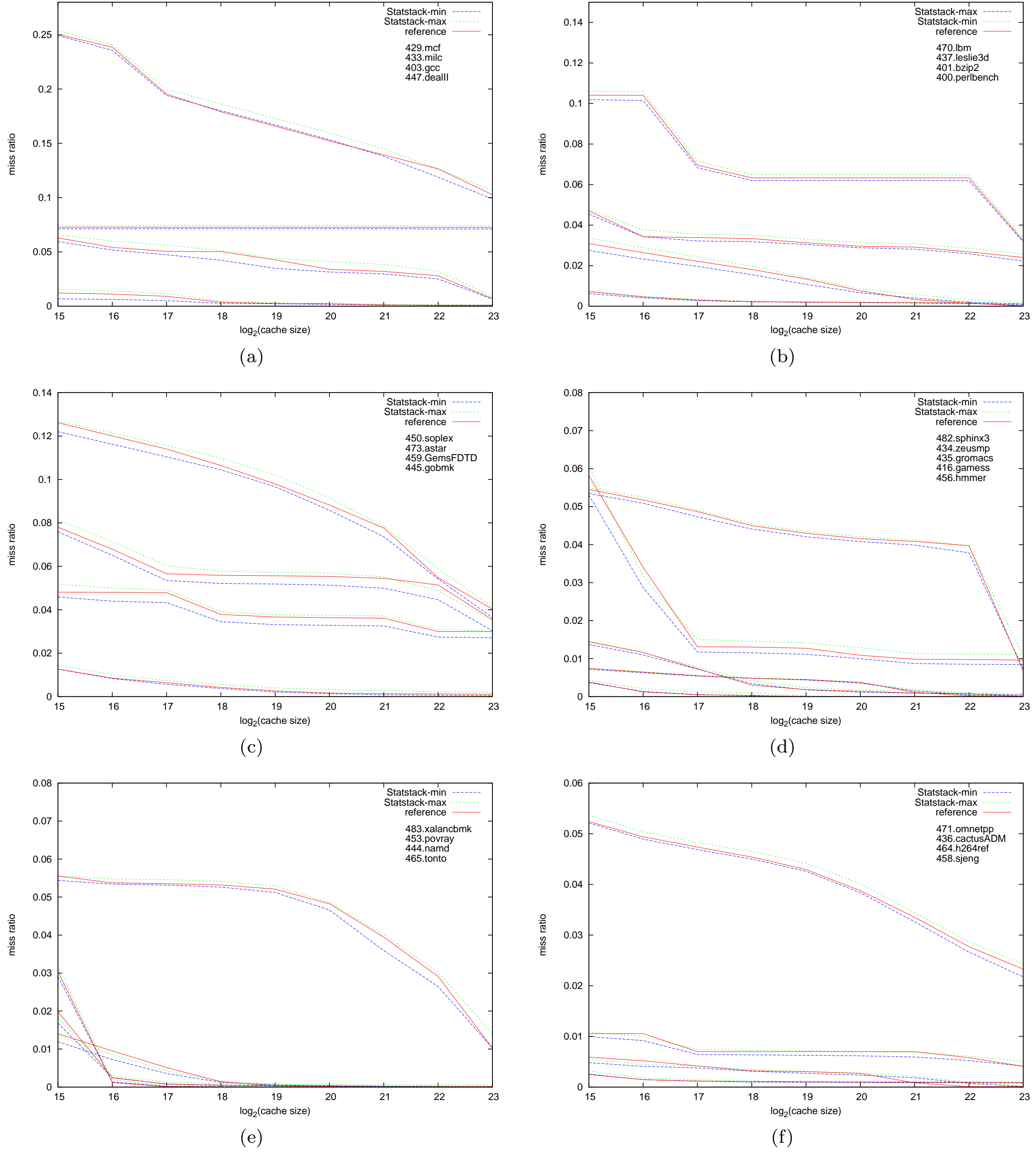

Figure 6: Miss ratios as a function of cache size. The graphs labeled Statstack-min and Statstack-max shows the minimum and maximum of 32 miss ratios estimated using 32 different RDS:s of size $N = 500,000$. The graph labeled Reference shows the reference miss ratio. The graphs are ordered by their y intercept according to the list on the right.

been proposed [17][8]. These approximate algorithms all use the full address trace as input.

A variety of sampling techniques have been proposed to improve the efficiency of the stack distance algorithm by reducing the input size and/or by reducing the overhead of the data collection. The two main approaches are set sampling [6][13] and trace sampling [14][13][20].

In set sampling only a small portion of the sets in a set associative cache are simulated. Set sampling has been used for decision making in adaptive cache replacement policies [16][9].

In trace sampling a set of sub address traces are randomly selected and collected from the target applica-

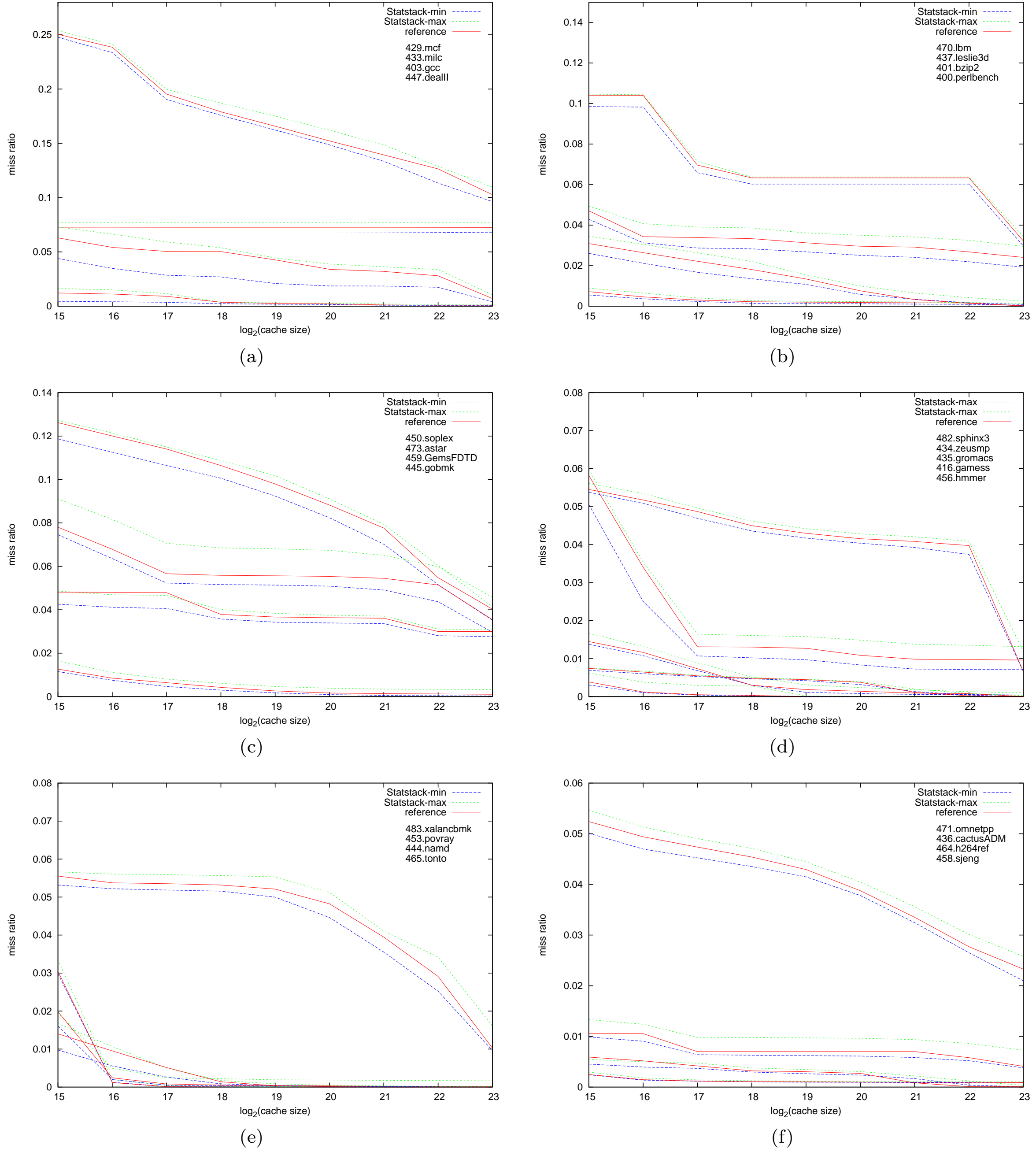

Figure 7: Miss ratios as a function of cache size. The graphs labeled Statstack-min and Statstack-max shows the minimum and maximum of 32 miss ratios estimated using 32 different RDS:s of size $N = 100,000$. The graph labeled Reference shows the reference miss ratio. The graphs are ordered by their y intercept according to the list on the right.

tion. The sub traces are run through a cache simulator and the overall miss ratio is estimated as the average of the sub traces' miss ratios. A drawback of this approach is that it requires the sub traces to contain a large number of memory accesses used only to warm the cache before the real simulation can start. The number of memory accesses needed to warm the caches and techniques to reduce this number have been investigated [19][11][10].

Shen et al. [17] presents a statistical cache model, where the input is a distribution of reuse distances and the working set size of the target application. It appears as their model can use an RDS to approximate the reuse distance distribution, and the number of dan-

gling samples to approximate the working set, but this is not considered.

8 Conclusions

This paper presented Statstack, a new statistical cache model that models a fully associative cache with LRU replacement policy. The input to Statstack cache model is an RDS, which contains the reuse distances of a sparse set of randomly selected memory references. To an obtain RDS:s, Statstack uses the same sampling technique as Statcache, and therefore inherits Statcache low data collection overhead.

The accuracy of Statstack has been evaluated using the SPEC CPU2006 benchmarks. The results show that Statstack can estimate accurate miss ratios based on RDS:s containing as few as 100,000 and 500,000 reuse distances. To produce these results, the execution time of the cache model is less than a few seconds.

References

[1] G. Almási, C. Caşcaval, and D. A. Padua. Calculating stack distances efficiently. *SIGPLAN Not.*, 38(2 supplement):37–43, 2003.

[2] B. T. Bennett and V. J. Kruskal. Lru stack processing. *IBM Journal of Research and Development*, pages 353–357, 1975.

[3] E. Berg and E. Hagersten. StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2004)*, Austin, Texas, USA, Mar. 2004.

[4] E. Berg and E. Hagersten. Fast Data-Locality Profiling of Native Execution. In *Proceedings of ACM SIGMETRICS 2005*, Banff, Canada, June 2005.

[5] E. Berg, H. Zeffer, and E. Hagersten. A Statistical Multiprocessor Cache Model. In *Proceedings of the 2006 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2006)*, Austin, Texas, USA, Mar. 2006.

[6] T. M. Conte, M. A. Hirsch, and W. mei W. Hwu. Combining trace sampling with single pass methods for efficient cache simulation. *IEEE Trans. Comput.*, 47(6):714–720, 1998.

[7] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 217, Washington, DC, USA, 2003. IEEE Computer Society.

[8] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. *SIGPLAN Not.*, 38(5):245–257, 2003.

[9] H. Dybdahl, P. Stenstrm, and L. Natvig. A cache-partition aware replacement policy for chip multiprocessors. In *In Proceedings of 13th International Conference of High Performance Computing (HiPC*, 2006.

[10] L. Eeckhout, S. Niar, and K. D. Bosschere. Optimal sample length for efficient cache simulation. *J. Syst. Archit.*, 51(9):513–525, 2005.

[11] L. V. Ertvelde, F. Hellebaut, L. Eeckhout, and K. D. Bosschere. Nsl-blrl: Efficient cachewarmup for sampled processor simulation. In *ANSS '06: Proceedings of the 39th annual Symposium on Simulation*, pages 168–177, Washington, DC, USA, 2006. IEEE Computer Society.

[12] C. Fang, S. Carr, S. Onder, and Z. Wang. Instruction based memory distance analysis and its application. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 27–37, Washington, DC, USA, 2005. IEEE Computer Society.

[13] R. E. Kessler, M. D. Hill, and D. A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Trans. Comput.*, 43(6):664–675, 1994.

[14] S. Laha, J. H. Patel, and R. K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Trans. Comput.*, 37(11):1325–1336, 1988.

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

[16] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. *SIGARCH Comput. Archit. News*, 35(2):381–391, 2007.

[17] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. *SIGPLAN Not.*, 42(1):55–61, 2007.

[18] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 23(6):84–93, 2003.

[19] D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In *SIGMETRICS '91: Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 79–89, New York, NY, USA, 1991. ACM.

[20] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. *SIGARCH Comput. Archit. News*, 31(2):84–97, 2003.

[21] Y. Zhong, S. G. Dropsho, X. Shen, A. Studer, and C. Ding. Miss rate prediction across program inputs and cache configurations. *IEEE Trans. Comput.*, 56(3):328–343, 2007.

A

Let l_t denote the cache line sized piece of memory accesses by memory reference x_t. We can now write Bennett and Kruskal's equations [2] as follows,

$$B_t(i) = \begin{cases} 1 & \text{if } l_i = l_j \text{ for } j = i+1, ..., t, \\ 0 & \text{otherwise} \end{cases}$$

$$P_t(x_k) = \begin{cases} \max{(i \leq t)} & \text{such that } l_i = l_k, \\ -1 & \text{otherwise} \end{cases}$$

$$S(x_t) = \sum_{i=p+1}^{t-1} B_{t-1}(i), \text{ where, } p = P_{t-1}(x_t) \quad (7)$$

To show that (1) follows from the above equations, we indentify the following equalities, $B_t(i) = 1(\vec{R}(x_i) + i > t)$ and $P_{t-1}(x_t) = t - R(x_t) - 1$. We now substitute these equalities into (7) and (1) follows.