A Statistical Multiprocessor Cache Model

Erik Berg, Håkan Zeffer and Erik Hagersten
Uppsala University, Department of Information Technology,
P.O. Box 337, SE-751 05 Uppsala, Sweden
{erikberg,zeffer,eh}@it.uu.se

Abstract

The introduction of general-purpose microprocessors running multiple threads will put a focus on methods and tools helping a programmer to write efficient parallel applications. Such a tool should be fast enough to meet a software developer's need for short turn-around time, but also be accurate and flexible enough to provide trend-correct and intuitive feedback.

This paper presents a novel sample-based method for analyzing the data locality of a multithreaded application. Very sparse data is collected during a single execution of the studied application. The architectural-independent information collected during the execution is fed to a mathematical memory-system model for predicting the cache miss ratio. The sparse data can be used to characterize the application's data locality with respect to almost any possible memory system, such as complicated multiprocessor multilevel cache hierarchies. Any combination of cache size, cache-line size and degree of sharing can be modeled. Each modeled design point takes only a fraction of a second to evaluate, even though the application from which the sampled data was collected may have executed for hours. This makes the tool not just usable for software developers, but also for hardware developers who need to evaluate a huge memory-system design space.

The accuracy of the method is evaluated using a large number of commercial and technical multi-threaded applications. The result produced by the algorithm is shown to be consistent with results from a traditional (and much slower) architecture simulation.

1 Introduction

Data locality is central to modern computer designs. The widening gap between processor speed and memory latency has introduced the need for a deep hierarchy of caches. Thus, the performance of an application is to a large extent dependent on the amount of data locality the caches can exploit. Some data locality comes naturally from the way most programs are written and the way their data is allocated in memory. Compilers further try to create data locality by loop transformations and optimized data layouts. Different ways of writing a program and/or laying out its data may improve an application's locality even more. However, it is far from obvious how such a locality optimization can be achieved, especially since the optimizing compiler may have left the optimization job half done. Thus, efficient tools are needed to guide the software developers on their quest for data locality.

CPU chips running multiple threads will likely become more common as a result of the introduction of multi-core chips. This motivates new methods for understanding the performance of systems built from such chips. This paper introduces a novel method for analyzing data cache behavior and communication patterns in a computer system built from CPUs with multiple cores.

We have in previous papers presented the statistical cache model StatCache. StatCache is a technique for estimating capacity misses of a single-processor computer [3, 4]. StatCache is based on sparse sampling and can produce accurate results at a very low sampling rate. The low sampling rate, in combination with a natural mapping between the properties sampled and the available functionality in modern CPUs and operating systems, enables efficient implementations.

In this paper we present methods for extending StatCache to model cold misses as well as to handle a system consisting of multiple processors connected in a symmetric or semi-symmetric configuration to a common memory, i.e., a SMP, NUMA or a chip multi processor (CMP). The new StatCacheMP framework presented can differentiate between misses caused by cache capacity replecements, cold misses and most cache misses caused by the coherence protocol.

StatCacheMP is very flexible. The statistical data sampled during an execution are independent of the architectural parameters of the host computer where the data were collected, such as its cache size and cache-line size. The cache behavior of the application for a selection of cache parameters can then be determined off-line using a mathematical model.

The mathematical cache model can be used to model any possible design-point of a CMP by just changing a few pa-

rameters of the formula and then applying it to the relatively small set of sample points collected during the sampling. The model presented in this paper supports the design parameters: cache size, cache-line size and degree of sharing, i.e., the number of processors sharing a cache, to be varied when solving the mathematical formula. This way, the performance numbers of a new design point can be determined within a fraction of a second. While this flexibility may come handy for a hardware designer exploring a large design space, it can also be used to determine special characteristics of an application.

This paper first recaptures the theory of StatCache and then extends the original StatCache model by including cold misses and coherence misses. The evaluation section compares the estimated results from StatCacheMP with results from a more traditional (but slower) full-system simulator, while varying some cache memory parameters. The paper is ended with a discussion about possible implementation options and a conclusion.

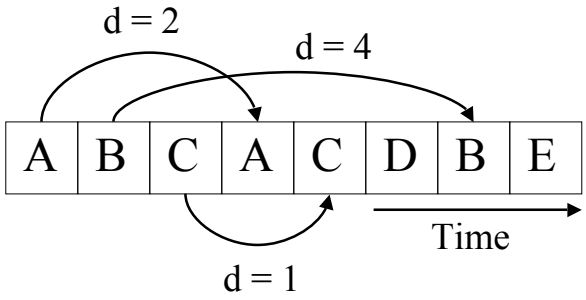

Figure 1. This figure illustrates the reuse-distance concept. Assume that the letters A, B, C, A, ... in the boxes represent cache-line-sized pieces of memory accessed in that order by a target application. The reuse distance, d, is the number of intervening accesses to other memory locations.

2 Related Work

Simulators are likely the most common method for evaluating hardware design tradeoffs. There are both commercial platforms and open-source software available for building simulators at different levels of detail. These range from slow timing-accurate simulators to fast but less accurate simulators [12, 16]. These simulators can be very accurate and/or very flexible. However, simulators are rarely used to give feedback to software developers because of their slow speed, which is not compatible with the short turn-around time used in software development. Furthermore, the setup of a simulation environment capable of running a complex application can often be cumbersome.

Simulation can be combined with sampling to reduce run-time overhead. Different approaches exist [8, 10, 11, 20]. The simplest possible scheme is to just run part of the application, that is, first warm the simulator and then run the experiment some fixed number of cycles, instructions or transactions. The drawback is that the samples may be very unrepresentative. Several research groups have addressed this issue. One way to overcome the problem is to pick several short samples and use well known statistical algorithms for finding the error bounds [21]. The experiment has to be repeated with a higher sampling rate if the error estimates are unacceptable. Another popular approach is to rely on phase detection algorithms in order to select the representative samples [15].

Static analysis is a completely different approach. It is based on one of the more well-known cache miss equations, such as the ones suggested by Martonosi [9] and others [7, 18]. Static analysis is a potentially very fast method. Static analysis can also be parameterized in terms of input data size etc. However, statistical analysis still requires some representative input of the characteristics of a execution. In order to capture these characteristics some simulation environment is often necessary.

The approach taken by StatCache can be viewed as a combination of sparse short samples and statistical analysis. However, StatCache can model large caches accurately even though sparse short samples are used.

3 Notation and Definitions

This section defines some commonly used symbols.

Reuse distance d:
Reuse distance, is central to both StatCache and StatCacheMP. Intuitively, reuse distance is the number of memory references between the current and previous memory reference to one specific cache-line-sized piece of memory. Figure 1 illustrates the concept. Note that all intermediate memory references are counted, not just different ones, as is the case for stack distance [14]. The reuse distance or stack distance can not easily be used to categorize the memory behavior as is, even though several attempts have been made [6, 5], but could be used for a more elaborate analysis to find cache miss ratios [22, 13].

Largest measured reuse distance d_{max}:
The largest measured reuse distance.

Total miss ratio:
$M_{total} = \frac{\#cache\ misses}{\#memory\ references}$

Read capacity miss ratio:
$M_{caprd} = \frac{\#load\ capacity\ misses}{\#memory\ references}$

Write capacity miss ratio:
$$M_{capwr} = \frac{\#store\ capacity\ misses}{\#memory\ references}$$

Cold miss ratio:
$$M_{cold} = \frac{\#cold\ misses}{\#memory\ references}$$

Read cold miss ratio:
$$M_{coldrd} = \frac{\#load\ cold\ misses}{\#memory\ references}$$

Write cold miss ratio:
$$M_{coldwr} = \frac{\#store\ cold\ misses}{\#memory\ references}$$

Read communication miss ratio:
$$M_{commrd} = \frac{\#load\ misses\ caused\ by\ remote\ invalidations}{\#memory\ references}$$

Write communication miss ratio:
$$M_{commwr} = \frac{\#store\ misses\ caused\ by\ remote\ invalidations}{\#memory\ references}$$

4 Capacity Misses: A StatCache Primer

This section recaptures the theory and implementation from the original StatCache proposal which is capable of estimating the capacity miss ratio of a single-processor computer.

4.1 The Basic Single Processor Model

Even though the reuse distances are hard to interpret as is, they can be transformed into cache-miss ratios using a statistical model. The basic StatCache model gives the miss ratio of a fully associative cache of arbitrary size with random replacement, as intuitively described here (please see [3] for a detailed description).

Assume that we know the reuse distance for all memory accesses. Then sort the reuse distances of all the memory accesses into histogram buckets, h_i. Let h_1 be the number of memory accesses with reuse distance one, h_2 the number of memory accesses with reuse distance two, and so on. We can then state that an equality equation for the total number of misses of the application: the average miss rate for the application (R) multiplied by number of memory references (N) is equal to the sum of the miss probability (m_i) for each bucket multiplied by the number of memory references of that bucket h_i, as shown in Equation 1.

$$R \cdot N = h_1 m_1 + h_2 m_2 + \ldots + h_{d_{max}} m_{d_{max}} \quad (1)$$

Assuming a fully associative cache with random replacement, the miss probability m_i can be calculated by the function $f(repl)$, which calculates the probability that a cache line has been evicted from a cache with L cache lines given that $repl$ replacements have occurred since we touched the cache line, as shown in Equation 2.

$$f(repl) = 1 - (1 - 1/L)^{repl} \quad (2)$$

Now, replacing m_i in Equation 1 with $f(repl)$ from Equation 2, where the number of replacements is approximated as the reuse distance times the miss probability R, yields the main equation of StatCache. This is shown by Equation 3, from which the miss ratio R easily can be solved numerically.

$$R \cdot N = h_1 f(R) + h_2 f(2R) + \ldots + h_{d_{max}} f(d_{max} R) \quad (3)$$

4.2 Sampling

Knowing the reuse distance for all accesses is a doable, but very in-efficient way of characterizing the behavior of an application. Fortunately, the shape of the histogram distribution $h(i)$ is easily approximated by sampling. It has been shown that the histogram distribution $h(i)$ for sparsely and randomly chosen samples of memory accesses is approximately equal to the shape of $h(i)$ based on every access [3].

Equation 3 is only valid if the miss ratio is approximately constant during the entire execution, which is not always the case. To handle this, StatCache divides the target-application run into several short periods, where each period is short enough for the miss ratio to be approximately constant. Such a period of the execution is called a *sampling window*. The sparse sample collected during each sample window is used as input to Equation 3 in order to estimate the miss ratio for that sample window and the overall miss ratio of the application is simply the arithmetic mean of the miss ratio of every sampling window.

4.3 An efficient implementation of StatCache

A prototype version of StatCache sampling, capable of taking StatCache samples from any unmodified application binary, exists and runs with a slowdown of about forty percent [4]. The execution of the application binary is halted about every $10^7 : th$ access by a user-level "spy process". The spy process will determine the address of a load or store performed by the application binary, set an operating system watchpoint for that address and determine the number of loads/stores performed since the application started by reading some hardware counter. A trap occurs the next time that address is touched and the hardware counter is read again in order to determine the reuse distance for that StatCache sample. More details of the method can be found elsewhere [4]

5 Cold Misses

The previous StatCache [3] proposal does not take cold misses into account. For short runs and/or large working sets, the cold misses become a significant fraction of all the misses and need to be handled in the equations. This section describes how to compute the cold miss ratio and how to extend StatCache to also handle cold misses.

5.1 Estimating the Cold Miss Ratio

A cold miss occurs when the processor accesses a cache-line-sized piece of memory that has never been touched before. Considering an application with a footprint of F cache-line-sized objects, each cache line will cause exactly one cold miss. Thus, the number of cold misses will be F. The cold miss ratio can be calculated as:

$$M_{cold} = \frac{\#cache\ lines\ in\ footprint}{\#memory\ references} \quad (4)$$

One way of estimating the footprint F would be to detect the first time each cache-line-sized object is touched and to count those events. Another way, which is more suitable for our sampling scheme, is to detect the *last* time each cache-line-sized memory object is touched and to count those events.

If we sample every memory access, the footprint F would correspond to the number of "dangling" samples at the end of the execution for which no reuse distance has been detected. If we instead sample with $sample\ rate$, the footprint F corresponds to $sample\ rate \times \#dangling\ samples$. Furthermore, the number of memory references would correspond to $sample\ rate \times \#samples$. This yields the equation:

$$M_{cold} = \frac{sample\ rate \times \#dangling\ samples}{sample\ rate \times \#samples} \quad (5)$$

$$M_{cold} = \frac{\#dangling\ samples}{\#samples} \quad (6)$$

5.2 Extending StatCache with Cold Misses

We need to factor in the impact of cold misses in Equation 3, which is only valid for capacity misses. First, let N_{nocold} be the number of warm memory references, i.e., the number of memory references that do not cause cold misses. Thus, $N_{nocold} = N(1 - M_{cold})$, where M_{cold} is the cold miss ratio. Next, use Equation 3 but refine the right-hand side estimate of the total number of cache misses by also adding the number of cold misses.

$$N \cdot R = N \cdot M_{cold} + h_1 f(R) + \ldots + h_{d_{max}} f(d_{max} R) \quad (7)$$

which is easily transformed into

$$\frac{N_{nocold}(R - M_{cold})}{1 - M_{cold}} = h_1 f(R) + \ldots + h_{d_{max}} f(d_{max} R) \quad (8)$$

Similarly to the original StatCache capacity miss formula, this formula assumes a constant miss rate throughout the application. Thus, the sample-window strategy we used in the original StatCache model is also needed in this cold-miss aware model.

6 Coherence Misses and Sharing

The StatCache model described so far can only estimate cache behavior for single-threaded execution. In this section we describe the extended sampling mechanism and the theory needed by StatCacheMP.

StatCacheMP models a MSI coherence protocol and classifies cache misses caused by invalidations as coherence misses. No upgrade misses are currently detected. While upgrade misses also could cause latency for an application, especially when run on a strong memory model, their detection would lead to a more complicated sampling model. Repeated upgrade misses to the same cache line will also most likely appear in concert with invalidation misses, which are detected by our model. We consider upgrade misses as future work.

6.1 More Detailed Sampling Description

Handling coherence misses and dividing the miss ratio into the categories capacity, cold and communication is a two step procedure. The first step is to estimate the overall miss ratio using the method described in Section 5.2 and the second step is to once again iterate through all the samples and estimate their respective cache miss probability. To enable the second step we must not only record the reuse distance histogram, but also record information about each individual sample. Note that the reuse distance is still considered to be a single processor (or possibly cluster of processors) metric, i.e., we keep monitoring the cache line until the same processor references the cache line again. The data we record for each sample is:

- The type of the first instruction, i.e., the instruction the sampling mechanism selects (load or store).
- The type of the second instruction, i.e., the instruction that causes a reuse to be detected.
- The reuse distance, d.
- The local processor ID, i.e., the ID of the processor executing the first instruction.
- Writer list: A list of the IDs of all other processors that write to the monitored cache line.

Given this information, the second step is straightforward. One simply iterates through all the samples and calculates their respective cache miss probabilities using Equation 2, $P_{miss} = f(d \cdot R_w)$, where d is the reuse distance of the sample, and R_w is the estimated miss ratio of the sampling window w that the sample belongs to. If the writer list contains a processor connected to another cache (another processor have written the data), the cache line was invalidated during the reuse distance measurement. However, this cache miss

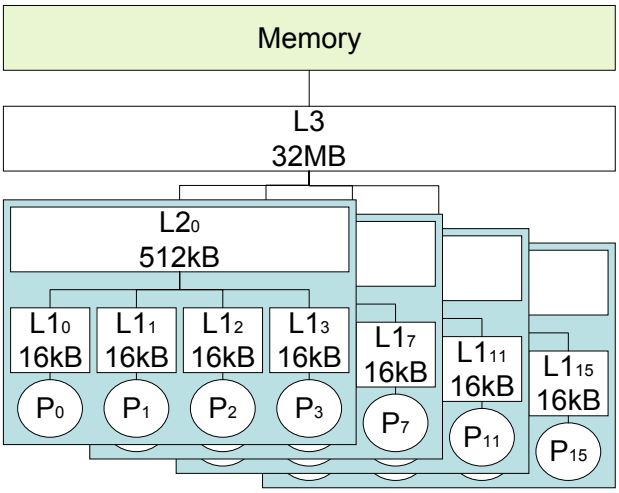

Figure 2. An example architecture configuration. Our multiprocessor cache model enables us to predict the miss ratio at all cache levels.

is only to be classified as a communication miss if the data would have resulted in a cache hit if no intervening write had occurred. Thus, an invalidated sample is deemed communication miss with the probability $1 - f(d \cdot R_w)$. Summing the respective miss probabilities of each individual sample and dividing by the number of samples estimate the final capacity and communication miss ratio. The samples may then be divided into categories according to whether the second instruction is a load or a store.

When creating sample data for several cache-line sizes from a single run, the reuse distance for each size must be monitored and a separate reuse histogram created for each cache-line size. Once a memory reference to the address A has been selected for reuse distance measurement, the reuse distance for the largest studied cache-line size containing A is monitored. Once a reuse for this cache-line size has been detected by an access to address B of this cache line, the histogram corresponding to the largest cache line is updated with this reuse value. Now we turn to the next largest cache-line size. If it can be determined that A and B also share the same cache line of the second largest cache-line size, its histogram is also updated with the reuse value. If not, the reuse distance for the second largest cache-line size containing A is measured, and so on.

6.2 Exploring the Multiprocessor Design Space

Similarly to single-processor StatCache, it is possible to use StatCacheMP to model several cache configurations based on the statistics collected from one execution. While StatCache allows different cache sizes and cache-line sizes to be modeled, StatCacheMP can also vary the degree of cache sharing in its post processing. Varying the degree of cache sharing requires a slight alteration to the capacity miss and cold miss equation so that all the samples from processors sharing a cache are taken into consideration for the calculation. Similarly, write accesses from another processor sharing the same cache will not cause coherence misses in the modeled system. (The ability to change configuration in terms of sharing is the reason why we collect a writer list and not simply mark a sample as invalidated.)

The StatCache and StatCacheMP equation only models one level of cache. However, if cache inclusion is assumed between the different levels of caches, the miss ratio of the different levels can be calculated independently. The same holds for different degrees of cache sharing at the different levels. Figure 2 shows one possible configuration for a CMP system with three levels of caches: sixteen 16kB private L1 caches, four 512kB shared by four L2 caches, and one 32MB shared by all L3 cache. One runs StatCacheMP on a selected benchmark and calculate separate curves for 1-, 4- and 16-node systems using StatCacheMP's post processing model in order to estimate the cache miss ratios for all cache levels in the example architecture. The miss ratio of the private first-level caches can be found in the 16-node data where the cache size is 256kB (the sum of all private L1 caches). The miss ratio of the second-level caches is found by dividing the 4-node data where the cache size is 2MB (the sum of all L2 caches) with the first-level cache's miss ratio. The first-level cache acts like a filter for the second-level cache. The miss ratio of the third-level cache can be calculated in a similar way.

7 Evaluation

This section evaluates the accuracy of the proposed algorithm for estimating the behavior of multiprocessor memory systems. Our earlier experiences from efficient StatCache implementations identified the importance of picking samples completely independent of each other, as well as sampling methods that overcome some of the difficulties imposed by modern CPU architectures. In order to isolate all such effects from the evaluation of the new mathematical algorithm presented in this paper, we have chosen a simulation-based evaluation methodology.

7.1 Simulation Methodology

Correctly interleaved memory reference traces are generated using the Simics full-system simulator [12]. Simics is configured to simulate a symmetric multiprocessor UltraSPARC-based system with 16 processors running Solaris. We use the traces to drive our reference cache simu-

lator as well as to collect the reuse distance samples fed to the multiprocessor-enhanced StatCache model.

The reference simulator models a single-level cache system. The processors in the reference simulator can either be connected in a symmetric configuration where each processor has its own data cache or connected in a non-uniform configuration where the processors are clustered in nodes where all processors in the same node share a common data cache. The caches are kept coherent using an MSI coherence protocol. The simulated caches are fully associative and implements random replacement. We have run the simulator with cache sizes from 2k byte to 64M byte and cache-line sizes from 32 bytes to 512 bytes.

We use a sampling rate of $1/5000$ in all experiments. This sampling rate gives between ten thousand and 25 thousand samples per thread for each benchmark. Our earlier StatCache work has shown that this amount of sample data should be enough to get reasonably good accuracy.

7.2 Benchmarks

APACHE: Static Web Content Serving: We use Apache 2.0.43 configured to use a hybrid multi-process multi-threaded server model with 64 POSIX threads per server process [1]. Our experiments use a hierarchical directory structure of 80,000 files (with a total data size of approximately 1.8 GB) and a modified version of the Scalable URL Reference Generator (SURGE [2]) to simulate 6400 users (400 per processor) with an average think time of 12ms.

SPECjbb2000: SPECjbb2000 is a server-side Java benchmark that models a 3-tier system, focusing on the middle-ware server business logic and object manipulation [17]. The benchmark includes driver threads to generate transactions as well as an object tree working as a back-end. Our experiment use 24-driver threads (1.5 per processor) and 24-warehouses (with a total data size of approximately 500MB).

SPLASH-2: We have also chosen to study a subset of the well-known workloads from the SPLASH-2 benchmark suite [19]. The selected programs were chosen to represent a variety of communication and synchronization requirements. For example, we use `fft` because of its communication-intensive behavior and `lu-nc` because it shows a lot of false sharing at large cache-line sizes.

Data set sizes for the applications studied can be found in Table 1. All SPLASH-2 applications are compiled with gcc-3.4.3 (optimization level 3). PARMACS macros for locks and barriers are based on user-level *test&test&set* spin locks. Pause/Event macros are implemented with the POSIX Pthread library (only `radix` uses a small amount of pauses).

Program	Problem Size
`apache`	1000 transactions
`fft`	256k points
`jbb`	5000 transactions
`lu-c`	512×512 matrices, 16×16 blocks
`lu-nc`	512×512 matrices, 16×16 blocks
`radix`	4M integers, radix 1024
`water-nsq`	512 molecules, 3 time steps
`water-sp`	512 molecules, 3 time steps

Table 1. Working set sizes of our benchmarks.

7.3 Breakdown comparison

Both the reference simulator and the StatCacheMP model produce the following miss ratios: total miss ratio, read and write capacity miss ratio, read and write cold miss ratio, and finally, read and write communication miss ratio.

The diagrams in Figure 3 show the cache-miss-ratio breakdown. Each diagram shows the breakdown for all benchmarks, the difference between the diagrams is the cache-line size and the cache size. We vary the cache-line size from 32 bytes to 512 bytes and presents results for the cache size 1M bytes and 16M bytes. Two bars are shown for each benchmark in each diagram, the left bar shows the output from the model, StatCacheMP, and the right bar shows the output from our reference simulator, RefSim. Each bar is divided into the cache miss types described above.

The results clearly show that StatCacheMP is able to identify the benchmarks with the largest amount of communication misses and also presents a rough estimate for the miss-ratio break down. It is easy to identify dominating cache-miss types, like cold misses or communication misses and it is also easy to study how communication varies with the cache-line size. Compare for example the cache-miss-breakdown diagrams for `apache`, when the cache line grows from 32 bytes to 512 bytes. StatCacheMP models in this case the increasing read communication with a promising accuracy and we believe that such analysis should prove useful for many purposes.

7.4 Cache- and Line-Size

StatCache allows for a fast way of exploring the miss ratio as a function of cache size since the cache size is a parameter in the model. It is therefore natural to evaluate the accuracy of StatCacheMP by plotting the miss ratio estimates of our model in the same graph as the reference simulator. The closer the curves are, the better is the accuracy. Figure 4 and 5 show such graphs for all our benchmarks when the cache size varies between 2kB and 16MB with the cache-line sizes 32, 128 and 512 bytes.

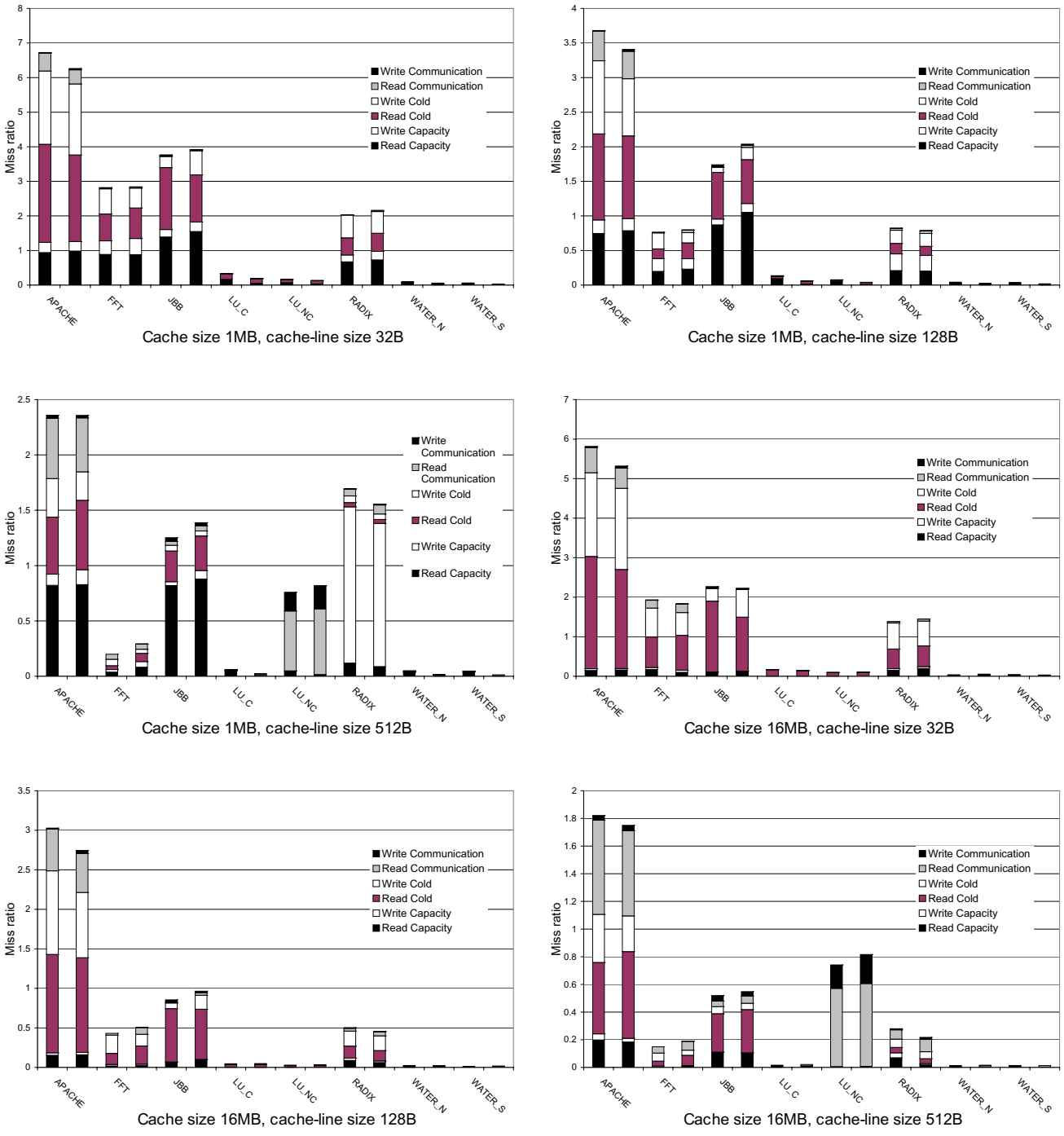

Figure 3. Miss ratio breakdown. For each benchmark, the left bar shows the StatCacheMP result and the right bar shows the reference simulator result. 16-processor, 4-node data.

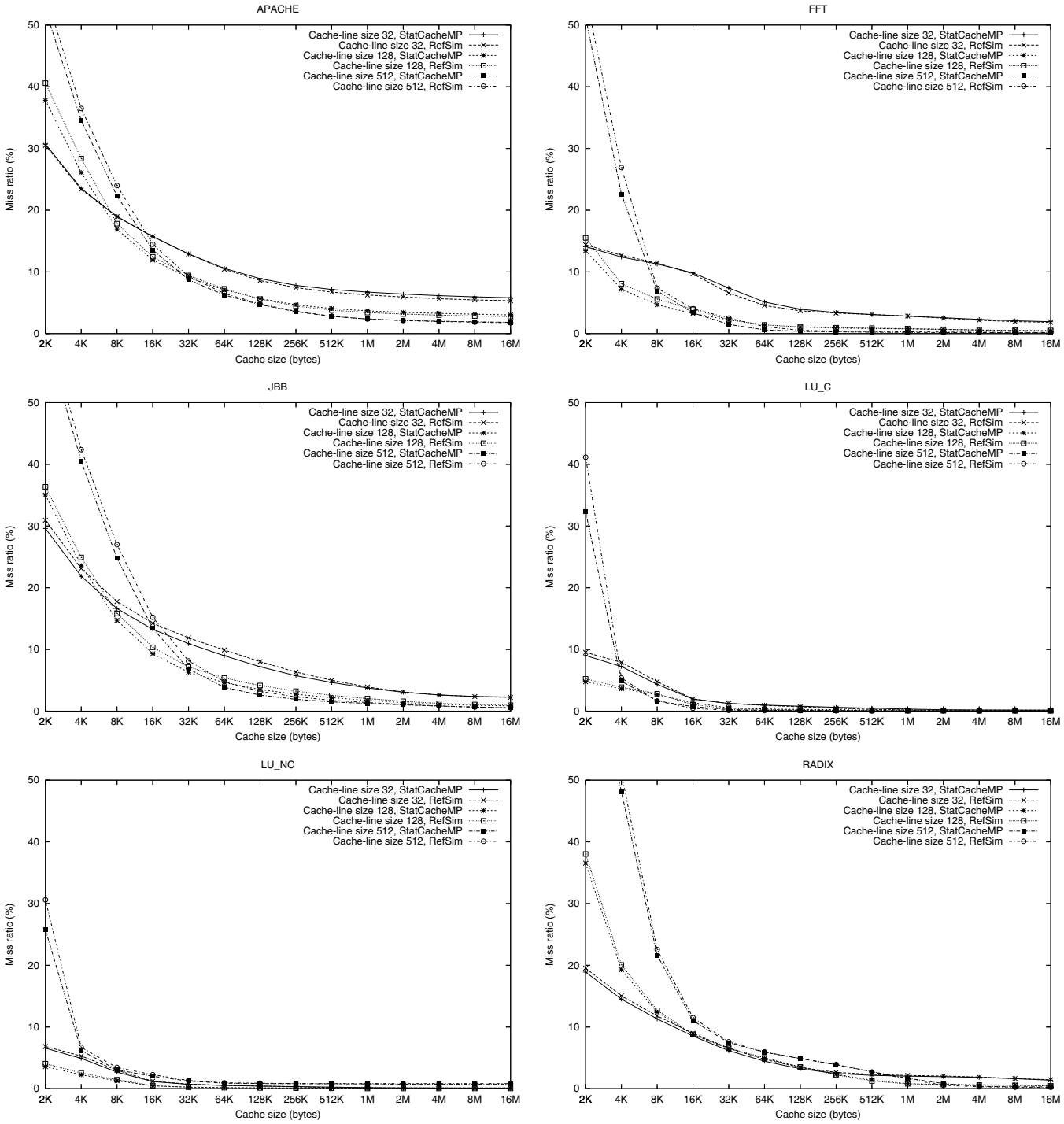

Figure 4. Miss ratio comparison of StatCacheMP and the reference simulator for `apache`, `fft`, `jbb`, `lu-c`, `lu-nc` and `radix` when the cache size is varied from 2kB bytes to 16MB. The cache-line sizes are 32, 128 and 512 bytes. 16-processor, 4-node data.

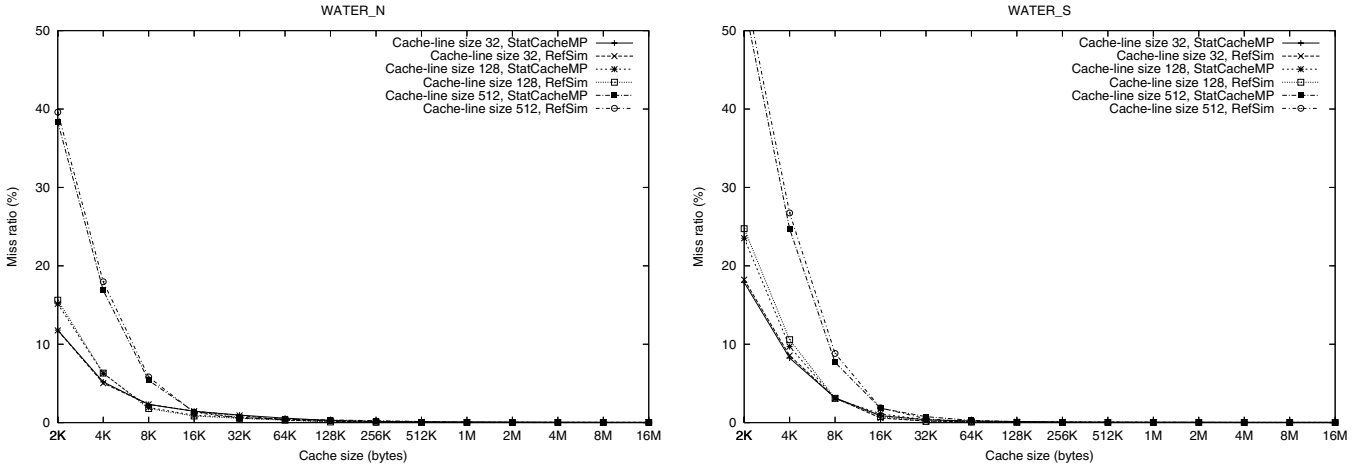

Figure 5. Miss ratio comparison of StatCacheMP and the reference simulator for `water-n` and `water-s` when the cache size is varied from 2kB to 16MB. The cache-line sizes are 32, 128 and 512 bytes. 16-processor, 4-node data.

StatCacheMP captures general trends very well. The curves of StatCacheMP follow the curves of the reference simulator very closely. The slopes are correct and the intersections between the curves for different line sizes are located at approximately the same places. The cold miss ratio estimated by StatCacheMP, i.e., the miss ratio of very large caches is also very close to that of the reference simulator. Note that the miss ratio is very large for very small caches and large cache-line sizes. The reason is that the cache contains very few cache lines. The extreme case is a 2kB cache with 512B cache lines. That cache configuration only has four cache lines, but StatCacheMP still estimates the miss ratio of such small caches well for most benchmarks.

An interesting example is `radix`. The knees of the curves are at different cache sizes for the different line sizes but StatCacheMP still produces results very similar to the reference simulator. This and similar examples strongly indicate that StatCacheMP handles a variety of complex memory reference patterns well, without loosing accuracy.

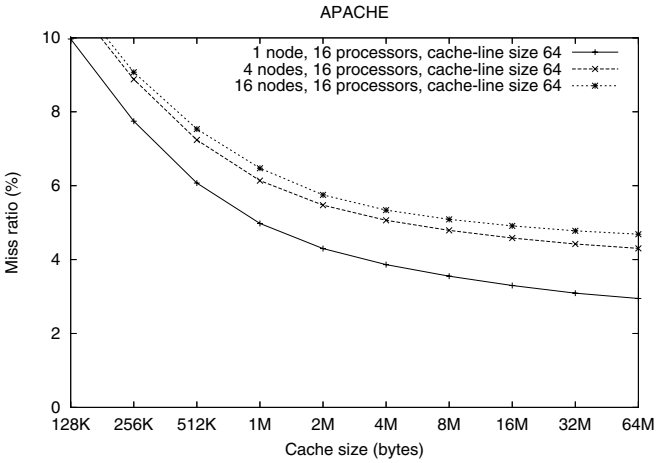

Figure 6. Miss ratio for `apache` while varying the degree of sharing in a multiprocessor system. 16-processor, 1-, 4- and 16-node data.

7.5 Sharing

Figure 6 and 7 show the miss ratio as a function of the cache size for the applications `apache` and `lu-nc`. Figure 6 (`apache`) shows three different configurations: 1 node means that all processors share a single cache, 4 nodes means that four processors share a cache and that the system has four caches in total, and 16 nodes means that each processor has its own private cache. All the configurations have a cache-line size of 64 bytes. We find it very interesting to see that a shared cache reduces the cache misses as much as they do for `apache`. We believe this is partly because of the big amount of time spent in the operating system.

Figure 7 (`lu-nc`) has the same node configurations as described for `apache` but includes two cache-line sizes: 128 and 256 bytes. The graph clearly shows that a cache-line size of 128 bytes is to prefer for the first-level cache and the second-level cache of our example architecture (see Figure 2). However, the third-level cache, which is shared between all the processors, does not suffer from false sharing and will benefit from a cache-line size of 256 bytes. Figure 8 shows in more detail that it is the false sharing that causes `lu-nc` to perform badly with multiple nodes and a cache-line size larger than 128 bytes. The figure shows

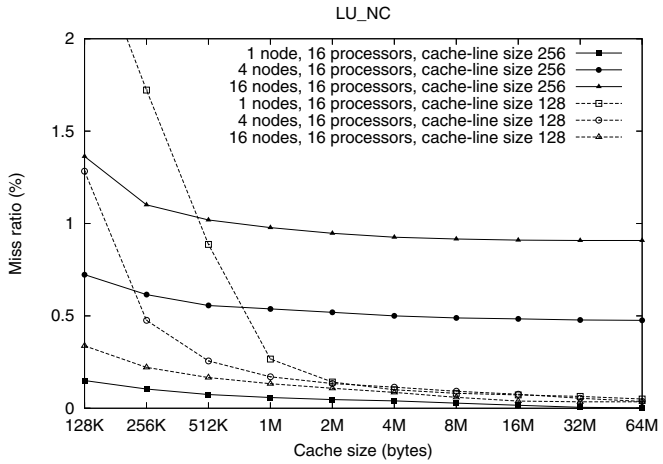

Figure 7. Miss ratio for `lu-nc` while varying the degree of sharing in a multiprocessor system. 16-processor, 1-, 4- and 16-node data.

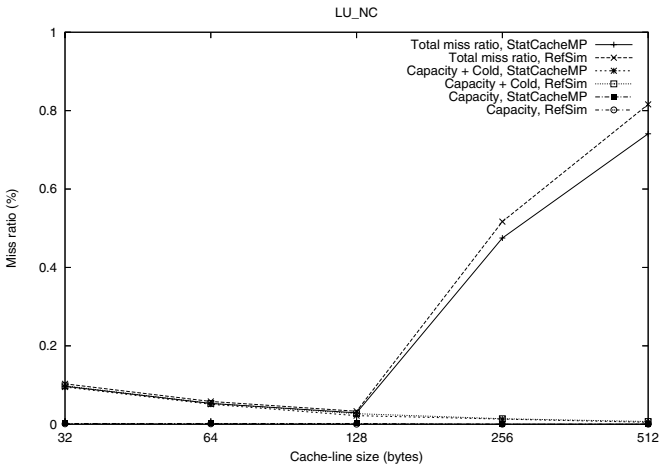

Figure 8. The miss ratio of `lu-nc` as a function of cache-line size. 16-processor, 4-node data.

the capacity, cold and communication misses plotted as a function of cache-line size. The miss ratio decreases for line sizes up to 128 bytes because of the decreasing number of cold misses and the extra spatial locality. For line sizes above 128 bytes, the miss ratio starts growing rapidly because of false sharing.

The graphs in Figure 6 and 7 can also be used to determine the miss ratio at each cache level in a memory system. The miss ratio of the first-level cache in our example architecture shown in Figure 2 is represented by the 16-node line for a cache size of 256kB (the cache size in the graph is the sum of all caches at that level), i.e., 9.1 percent for `apache`. The miss ratio for the second-level cache is read at the 4-node line for a cache size of 2 MB divided by the first-level cache miss ratio[1], i.e., 60 percent. In a similar way, the miss ratio of the third-level cache is 61 percent.

7.6 Result and Error Discussion

As all analytical models and sampling-based methods, there are a few possible error sources for StatCache. As always when it comes to sampling, it is very unlikely that the collected samples exactly represent the studied population.

The most important simplification in the model is likely the assumption that the miss ratio is constant in the sampling windows. If the sampling window is too long, a miss ratio that varies much is hard to estimate. A too short sampling window will cause poor results because each sampling window will contain too few samples to give a good approximation of the reuse-distance distribution. We have found the results to be pretty stable for window sizes between 30 and 300 samples.

[1] Assuming inclusion between the cache levels.

The handling of cold misses also introduces possible errors. Cold misses are not uniformly distributed across the application run. Instead, we get the average cold miss ratio for a period of time. This time period is ten times the sampling window in this implementation. The sampling rate is $1/5000$ and the sampling window size is 100 samples in our experiments, i.e., the cold miss ratio is assumed to be constant over a period of half a million memory references.

The breakdown of the categories capacity, cold and communication into the subcategories read and write assumes that the miss probabilities for memory read and write instructions are independent. In practice, read and write instructions are not independent; rather, they often appear in repeated patterns for example caused by loops.

StatCacheMP allows for an efficient exploration of a huge design space. However, it also introduces a possible source of errors in the estimations. A different cache configuration will most likely result in a different memory access interleaving. Hence, a slightly different amount of coherence misses might be encountered. While we believe that a program's sharing behavior rather is a property of the program than a property of the latencies in a certain configuration, this is a very important and interesting study we plan to address as future work.

This paper does not include any error estimates. However, the absolute values in the breakdown graphs are overall rather accurate in comparison with the reference simulator. StatCacheMP also captures important trends, such as the amount of false sharing as a function of cache-line size. Of course there might be pathological cases where StatCacheMP might not work, but we have not come across any such case.

8 Implementation Discussion

We have previously presented a prototype implementation with a run-time overhead of about forty percent, but believe that a full-fledged implementation with operating system support will have an overhead of only a few percent. The prototype can collect sample data from complex applications running natively on the host. The sampler is implemented as a background process monitoring the studied application.

The extended sampling mechanism needed to extend the original StatCache proposal for modeling multiprocessor systems should not add much more overhead. The only difference is that the watchpoint must be thread-global.

9 Conclusion

This paper describes a flexible and fast way of modeling an application's behavior on a multiprocessor memory hierarchy. Based on sparse architecturally independent data collected from a single multithreaded execution, a plentiful of different multiprocessor memory systems can be modeled using a mathematical model. The number of cache levels, their respectively cache size and cache-line size as well as the degree of cache sharing at each level can all be chosen arbitrarily when the mathematical formula is solved. The performance estimate for each such design-point takes fractions of a second to calculate, even though the actual application studied may take hours to run in a production environment.

The performance estimates produced by the model are broken up into six different miss categories. We show that the results obtained are fairly accurate in spite of the flexibility and speed provided by the method.

10 Acknowledgments

We would like to thank David Wood and Mark Hill at University of Wisconsin for providing us with the `apache` and the `jbb` benchmarks. This work is supported by Sun Microsystems, Inc. Uppsala Architecture Research Team (UART) is a member of the HiPEAC network.

References

[1] The Apache Software Foundation. *Apache HTTP Server Version 2.0.43 Reference Manual*.

[2] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *SIGMETRICS'98*, 1998.

[3] E. Berg and E. Hagersten. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *ISPASS'04*, 2004.

[4] E. Berg and E. Hagersten. Fast Data-Locality Profiling of Native Execution. In *SIGMETRICS'05*, 2005.

[5] K. Beyls et al. Visualization enables the programmer to reduce cache misses. In *PDCS'02*, 2002.

[6] K. Beyls et al. RDVIS: A Tool that Visualizes the Causes of Low Locality and Hints Program Optimizations. In *ICCS'05*, 2005.

[7] C. Cascaval and D. A. Padua. Estimating Cache Misses and Locality using Stack Distances. In *ICS'03*, 2003.

[8] T. M. Conte et al. Combining Trace Sampling with Single Pass Methods for Efficient Cache Simulation. *IEEE Transactions on Computers*, 47(6), 1998.

[9] S. Ghosh et al. Cache Miss Equations: A Compiler Framework for Analyzing and tuning Memory Behavior. *ACM Transactions on Programming Languages and Systems*, 21(4), 1999.

[10] R. E. Kessler et al. A Comparison of Trace-Sampling Techniques for Multi-Megabyte Caches. *IEEE Transactions on Computers*, 43(6), 1994.

[11] S. Laha et al. Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems. *IEEE Transactions on computers*, 1988.

[12] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2), 2002.

[13] G. Marin and J. Mellor-Crummey. Cross-Architecture Performance Predictions for Scientific Applications Using Parameterized Models. In *SIGMETRICS'04*, 2004.

[14] R. L. Mattson et al. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2), 1970.

[15] E. Perelman et al. Using SimPoint for Accurate and Efficient Simulation. In *SIGMETRICS'03*, 2003.

[16] M. Rosenblum et al. Using the SimOS Machine Simulator to Study Complex Systems. *ACM Transactions on Modelling and Computer Simulation*, 7, 1997.

[17] Standard Performance Evaluation Corporation. *SPECjbb2000, A Java Business Benchmark*. White Paper.

[18] X. Vera and J. Xue. Let's Study Whole-Program Cache Behaviour Analytically. In *HPCA-8*, 2002.

[19] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA-22*, 1995.

[20] D. A. Wood et al. A Model for Estimating Trace-Sample Miss Ratios. *ACM SIGMETRICS Performance Evaluation Review*, 19(1), 1991.

[21] R. E. Wunderlich et al. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *ISCA-30*, 2003.

[22] Y. Zhong et al. Miss Rate Prediction across All Program Inputs. In *PACT'03*, 2003.