A Statistical Multiprocessor Cache Model

Erik Berg, Håkan Zeffer and Erik Hagersten

Department of Information Technology
Uppsala University
Box 337, SE-751 05 Uppsala, Sweden

Technical report 2005-028
November 2005
ISSN 1404-3203

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 describes an efficient and flexible approach for modeling the memory system of a multiprocessor, such as those of chip multiprocessors (CMPs). Sparse data is sampled during a multithreaded execution. The data collected consist of the reuse distance and invalidation distribution for a small subset of the memory accesses. Based on the sampled data from a single run, a new mathematical formula is used to estimate the miss rate for a multiprocessor memory hierarchy built from caches of arbitrarily size, cache-line size and degree of sharing. The formula further divides the misses into six categories to aid the software developer.

The method is evaluated using a large number of commercial and technical multi-threaded applications. The result produced by our algorithm fed with sparse sampling data is shown to be consistent with results gathered during traditional architecture simulation.

1 Introduction

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. One of the most important performance aspects of a modern computer is the cache behavior. This report 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 uni-processor computer [1, 2]. 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 system, enables efficient implementations.

StatCache is also 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 cache size and cache-line size. The cache behavior of the application for a selection of cache parameters can be determined off-line using a mathematical model for caches. Thus, the cache parameters, such as cache-line size and cache size, may be varied during the post-processing analysis to investigate for example spatial locality properties of an application.

We have previously presented a prototype implementation with a run-time overhead of about forty percent, but believe a full-fledged implementation 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.

In this report 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 multi-chip processor (CMP). The new StatCacheMP framework presented here can differentiate between misses caused by cache capacity replacement, cold misses and cache misses caused be the coherence protocol actions.

This report first recaptures the theory of StatCache and 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 simulation, while varying some cache memory parameters. The report is ended with a discussion about possible implementation options and a conclusion.

2 Related Work

Simulators are likely the most common method for evaluating hardware design trade offs. There are both commercial platforms and open software available for building simulators at different levels of detail. These range from the slower timing-accurate simulators to the less slow functional simulators [10, 14]. 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 [6, 8, 9, 16]. The simplest possible is just running 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. This issue has been addressed by several research groups. One way to overcome the problem is to pick several short samples and use well-known statistical algorithms for finding the error bounds[17]. 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 [13].

Static analysis is a completely different approach. It is based on one of the more of the well-known cache miss equations, such as the ones suggested by Martonosi [7] and others [5, 15]. 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 the 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. Both the mathematical formulas used to model the system as well as the method used to collect the sampled properties of the system are new. The statistical equations used in StatCache rely on the property *reuse distance*.

3 Capturing Capacity Misses: A StatCache Primer

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

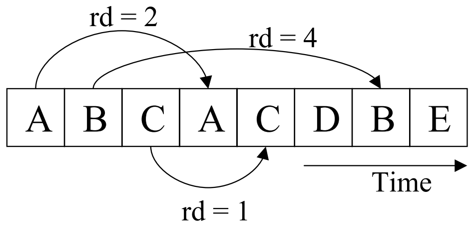


Figure 1: This figure illustrates the reuse-distance concept. Assume that the letters A, B, C, A, ... in the boxes represent cache-line-sized pieces of memory accessed in that order by a target application. The reuse distance, rd, is the number of intervening accesses to other memory locations. For example the first access goes to cache line A, and the next time A is accessed is in the fourth access. This gives the reuse distance two.

3.1 Reuse Distance

The reuse distance is central to both StatCache and StatCacheMP. Intuitively, the 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 [12]. 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 [3, 4], but could be used for a more elaborate analysis to find cache miss ratios [18, 11].

3.2 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 [1] 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_0 be the number of memory accesses with reuse distance zero, h_1 the number of memory accesses with reuse distance one, 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.

$$R \cdot N = h_1 M_1 + h_2 M_2 + h_3 M_3 + \dots \quad (1)$$

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 yields the main equation of StatCache, as shown by Equation 3, from which the miss ratio R easily can be solved numerically.

$$R \cdot N = h_1 f(R) + h_2 f(2R) + h_3(R3) + \ldots \qquad (3)$$

3.3 Sampling

Knowing the reuse distance for all accesses is a doable, but not a very 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 [1].

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 estimated 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 Cold Misses

The StatCache [1] proposal proposals to date do not take cold misses into account. For shorter run-times and/or large working sets, the cold misses become a more 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 the cold misses.

4.1 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. Thus, the number of cold misses will be equal to the data footprint measured at cache-line granularity. Dividing the footprint size with the total number of memory references gives the cold miss ratio, CMR, i.e., the ratio of memory references for which cold misses occur.

$$CMR = \frac{footprint\ size}{number\ of\ memory\ references \times cache - line\ size} \qquad (4)$$

Assume we are using a sampling scheme similar to StatCache. First, we pick random memory data references, find their effective addresses and the corresponding cache-line-sized piece of memory, and put a mark on that piece of memory so that we can later detect when the processor attempts to access the same piece of memory again.

Figure 2 illustrates the method for estimating the number of cold misses with a tiny example trace. The references are represented as pairs, (CL, I), where CL identifies the accessed cache-line-sized piece of memory (or simply cache line), i.e., the effective address divided by the cache-line size, and I is the reference counter (time). The references are sorted according to address and reference counter. Some of the references are sampled and monitored for reuse (marked by circles), but not all monitored cache

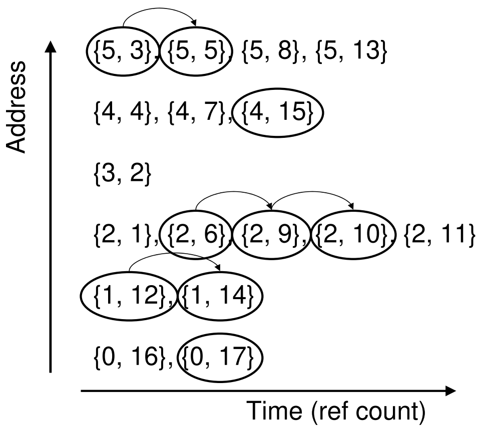


Figure 2: An illustration of the method we use for computing the cold miss ratio. The references are represented as pairs, (CL, I), where CL identifies the cache line, and I is the reference counter (time). The references $(5,3), (2,6), (2,9), (1,12), (4,15), (0,17)$ are sampled and monitored for reuse. The arrows point at the references where the cache lines are rereferenced. Some samples, i.e., $(4,15)$ and $(0,17)$ in the example, are the last references to the cache line and we can compute the cold miss ratio by counting how many such samples there are.

lines are ever reused. In the example these references are $(4, 15)$ and $(0, 17)$. At the end of the application run we can count how many monitored cache lines there are left. While the probability that the last reference to a cache line is sampled is equal to the sampling rate, it is easy to estimate the total number of different cache lines.

Let the sampling rate be S, i.e., samples are selected with probability S. Thus, the probability that the last reference to a cache-line-sized piece of memory is sampled is S. Let u be the number of samples such that reuse is never detected. The number of different cache-line-sized pieces of memory can be approximated with $S \cdot u$:

$$\#cold\ misses = S \cdot u. \tag{5}$$

The number of cold misses is the same as the footprint size divided by the cache line size, which gives us the cold miss ratio, CMR:

$$CMR = \frac{S \cdot u}{number\ of\ memory\ references}. \tag{6}$$

4.2 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. Thus, we need to remove the cold miss ratio for both sides of the equation:

$$N(R - CMR) = (1 - CMR)(h_1 f(R) + h_2 f(2R) + h_3 f(3R) + \ldots). \tag{7}$$

$$\frac{N(R - CMR)}{1 - CMR} = h_1 f(R) + h_2 f(2R) + h_3 f(3R) + \dots. \tag{8}$$

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.

5 Coherence Misses

The StatCache model described so far can only estimate cache behavior for single-threaded execution. In this section we describe the extended sampling mechanism needed by StatCacheMP, which also can model misses caused by the actions of a cache coherence protocol. We have chosen to model an MSI protocol. Cache misses caused by invalidation are counted as coherence misses only if the cache line otherwise would have remained in the cache at the time of the next local read or write access. No upgrade misses are detected. While upgrade misses also could cause latency for an application, especially for stronger memory models, 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 will invalidation misses, which are detected by our model.

In order to model the effects of the coherence protocol we need to distinguish between read and write miss ratio. This information can be extracted from the StatCache framework if the sampler records the type, read or write, of the second memory operation in every reuse-distance measurement.

While measuring the reuse distance between two accesses from one processor to a specific cache line, it is possible that a write access from a different thread is detected. If so, the reuse distance between the two local accesses is recorded but the access is marked as invalidated. If the reuse by itself would have resulted in a cache hit, the second local access is counted as a communication miss, otherwise a capacity miss is recorded.

5.1 More Detailed Sampling Description

While the simplistic description of the StatCache theory and sampling presented in Section 3.3 is sufficient for the basic StatCache equation, the additions and flexibility presented in this papers requires a few additions.

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.

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 4.2 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 for the post processing. 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 stop at. The type is load or store.
- The type of the second introduction, i.e., the instruction that causes a reuse to be detected.
- The local processor ID, i.e., the ID of the processor executing the first instruction.
- A writer list:A list of the IDs of all other processors that write to the monitored cache line.
- The reuse distance, d

Given this information, the second step is straight forward. Iterate through all the samples and calculate their respective cache miss probabilities using Equation 2, $P(cachemiss) = f(r \cdot d)$, where r is the estimated miss ratio of the sampling window the sample belongs to, and d is the reuse distance of the sample. If the writer list contains a processor connected to another cache, the cache line may have been invalidated during the reuse distance measurement. However, we do not want to count this as a communication miss unless the reuse distance is so short that the cache line would have remained in the cache if it had not been invalidated. Thus, a sample is deemed communication miss with the probability $1 - f(r \cdot d)$. The final capacity and communication miss ratio is estimated by summing the respective miss probabilities of each individual sample and dividing by the number of samples. The samples may then be divided into categories according to whether the second instruction is a load or store.

5.2 Exploring the Multiprocessor Design Space

Similarly to the uniprocessor StatCache, it is possible to use StatCacheMP to model several cache configurations based on the reuse-distance 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 when we use the mathematical formula to calculate the performance of a modeled system. Varying the degree of cache sharing requires a slight alteration to the capacity miss and cold miss equation so that all the samples from threads sharing a cache are taken into consideration for the calculation. Similarly, write accesses from another thread sharing the same cache will not cause coherence misses in the modeled system.

The StatCache and StatCacheMP equation only models one level of caching. If cache inclusion is assumed between the different levels of caching, the miss rate of the different levels of caches can be calculated independently. The same holds for different degree of cache sharing at the different levels.

While this allows for an efficient exploration of a huge design space, it also introduces a possible source of errors in the estimations. A different cache configuration could possible have resulted in a different execution order with a different amount of coherence misses. This source of error is similar to that of trace-driven simulation or functional simulation that does not model contention and latency in a correct way.

6 Evaluation

This section evaluates the accuracy of the proposed algorithm for estimating memory system behavior of multiprocessor memory system. Our earlier experiences from efficient StatCache implementations identified the importance of picking samples completely independently 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 trace-based evaluation methodology.

The trace-based evaluation enables us to pick samples randomly and measure the reuse distances without errors. It also has the advantage that we can feed exactly the same memory trace to both the reference simulator and the StatCacheMP sampling mechanism. This allows for easier comparisons between simulated and calculated performance numbers.

6.1 Evaluation setup

Memory reference traces are generated using the Simics full-system simulator. Simics is configured to simulate a symmetric multiprocessor UltraSPARC-based system running Solaris.

We use the traces to drive our reference cache simulator 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.

The StatCacheMP sampling study implements the necessary sampling mechanism and reuse-distance measurement mechanism. The sampling rate is set to 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. We use a configuration with 16 processors in total. These processors are divided in four nodes, where each node has its own cache. Thus, the number of caches in the modeled system is four.

6.2 Breakdown comparison

Both the reference simulator and the MPStatCache simulator produce
the following miss ratios:

- Total miss ratio: $M_{Total} = \frac{\#Cache\ misses}{\#Memory\ references}$
- Read capacity miss ratio: $M_{CapRd} = \frac{\#Load\ cache\ miss}{\#Memory\ references}$
- Write capacity miss ratio: $M_{CapWr} = \frac{\#Store\ cache\ miss}{\#Memory\ references}$
- Read cold miss ratio.: $M_{CdRd} = \frac{\#Load\ cold\ misses}{\#Memory\ references}$

- Write cold miss ratio: $M_{CdWr} = \frac{\#Store\ cold\ misses}{\#Memory\ references}$
- Read communication miss ratio:
$M_{CommRd} = \frac{\#Load\ misses\ caused\ by\ remote\ invalidations}{\#Memory\ references}$
- Write communication miss ratio:
$M_{CommWr} = \frac{\#Store\ misses\ caused\ by\ remote\ invalidations}{\#Memory\ references}$

The diagrams in Figure 3 (as well as in the Appendix A) 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 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 present 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.

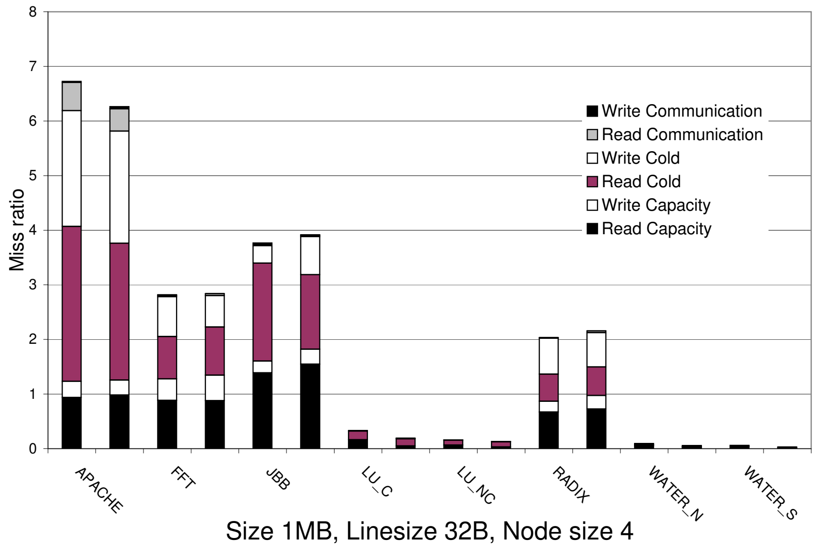
As all analytical models and sampling-based methods, there are a few error sources. This includes sampling error, the independence assumptions that are only approximately correct and other simplifying assumptions in the model. The first thing that comes to mind is sampling errors. As always when it comes to sampling, it is very unlikely that the collected samples exactly represent the studied population. Though we have not paid this problem much attention in this paper, it would be interesting to develop some method for estimating the sampling error. Estimates with error bounds are much more useful and trustworthy.

The next error source is the simplifying assumptions in the theoretical model. The most important simplification is likely the assumption that the miss ratio is constant in the sampling windows. It is a tradeoff weather to use a long or short sampling window. 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.

The results in this paper are generated with window size equal to 100 samples. We found this value by experimenting with different values and evaluating the results. The results are pretty stable for window sizes between 30 and 300 samples.

The handling of cold misses also introduces an error. 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. It



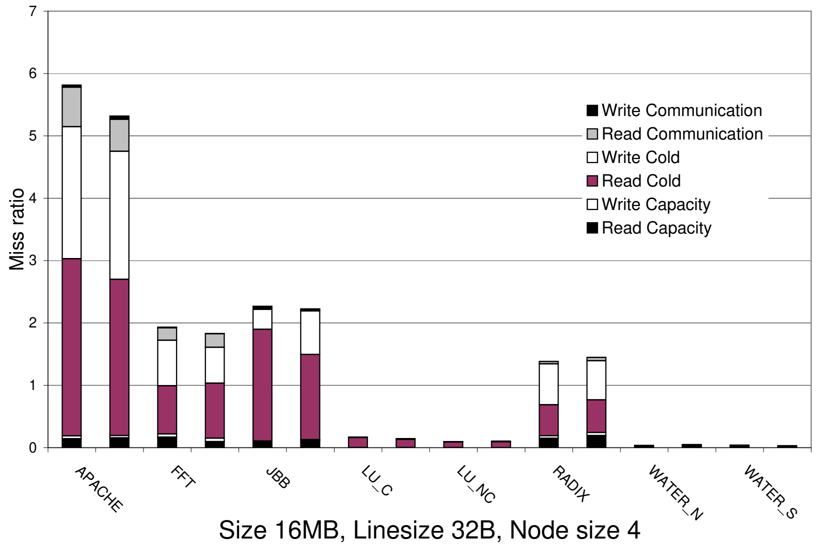


Figure 3: Miss ratio of the eight benchmarks broken down into the categories read capacity, write capacity, read cold, write cold, read communication and write communication. For each benchmark, the left bar shows the results from StatCacheMP and the right bar shows the results from the reference simulator.

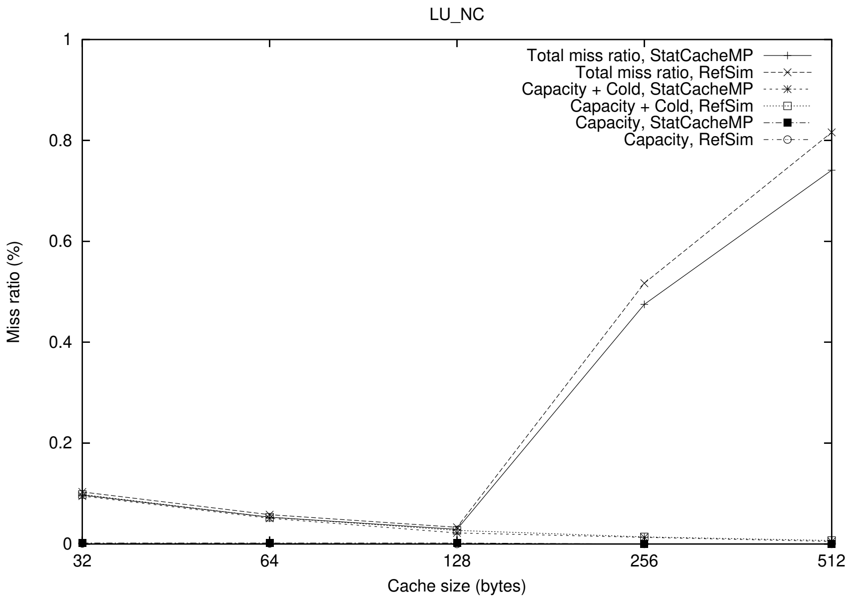


Figure 4: The miss ratio of LU_NC as a function of cache-line size. The total miss ratio decreases for line sizes up to 128 because of the decreasing number of cold misses. For line sizes above 128 bytes, the miss ratio starts growing rapidly because of false sharing. (Total miss ratio is the sum of cold, capacity and communication misses.)

could for example be that the read instructions all appear in program periods where the miss ratio is high and the write instructions appear when the miss ratio is low.

In spite of all the possible error sources, 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. Consider LU_NC as an illustrative example.

The communication miss ratio is low for cache-line sizes up to and including 128 bytes. Above this limit, the miss ratio grows rapidly.

Figure 4 shows this effect very clearly. The figure shows the capacity, cold and communication misses of LU_NC plotted as a function of cache-line size.

6.3 MR vs. size 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 5 shows such graphs for the APACHE benchmark for cache sizes between 2KB and 16MB, for the cache-line sizes 32, 128 and 512 bytes. The corresponding graphs for the other benchmarks in the evaluation are in Appendix B.

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

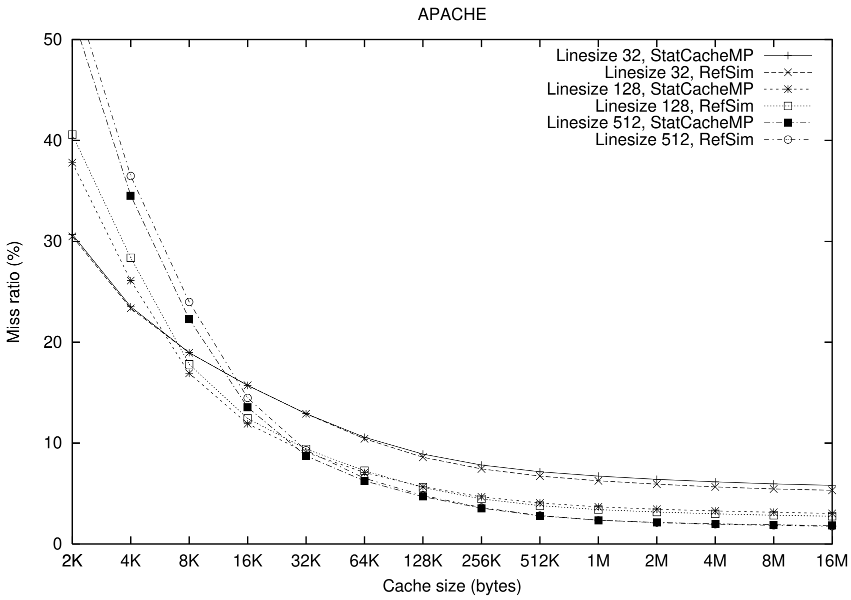


Figure 5: A comparison between StatCacheMP and the reference simulator. The cache size is varied from 2K bytes to 16M bytes. The line sizes are 32, 128 and 512 bytes. See also Appendix B.

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.

Another interesting example is RADIX (See App. B, Fig. B.5). The knees of the curves are at different cache sizes for all 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. Also in this case are the cold miss ratio estimates of StatCacheMP very close to the values presented by the reference simulator.

7 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.

Appendix A: Miss Ratio Breakdown

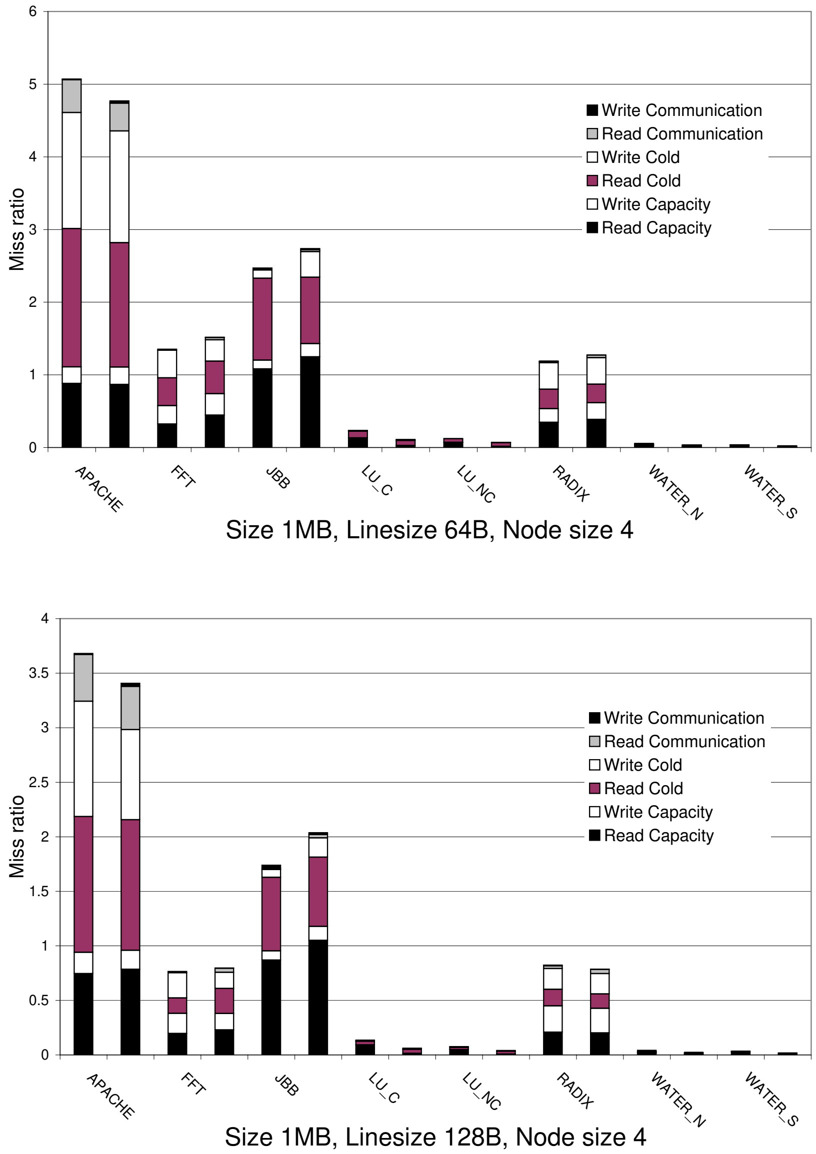
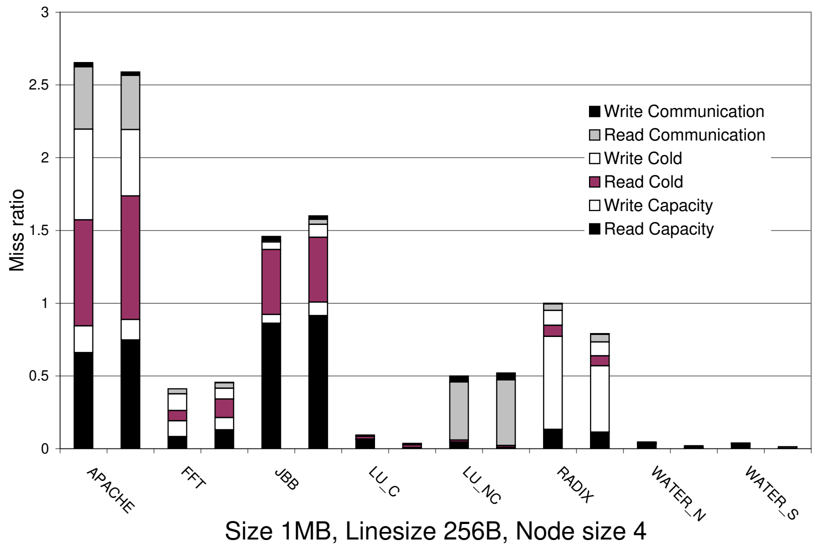


Figure 6: This diagrams show the miss ratio breakdown for the benchmarks APACHE, FFT, JBB, LU_C, LU_NC, RADIX, WATER_N and WATER_S. The cache size is 1M bytes and the cache-line size is 64 and 128 bytes.



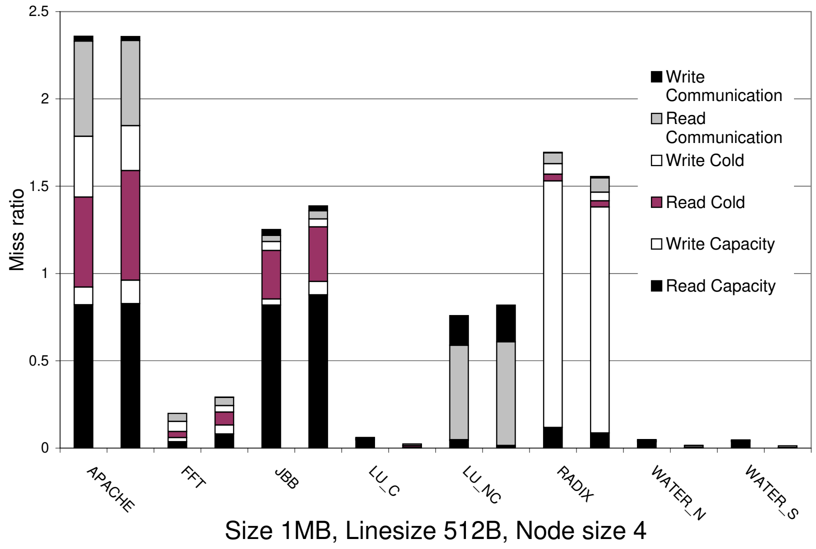


Figure 7: This diagrams show the miss ratio breakdown for the benchmarks APACHE, FFT, JBB, LU_C, LU_NC, RADIX, WATER_N and WATER_S. The cache size is 1M bytes and the cache-line size is 256 to 512 bytes.

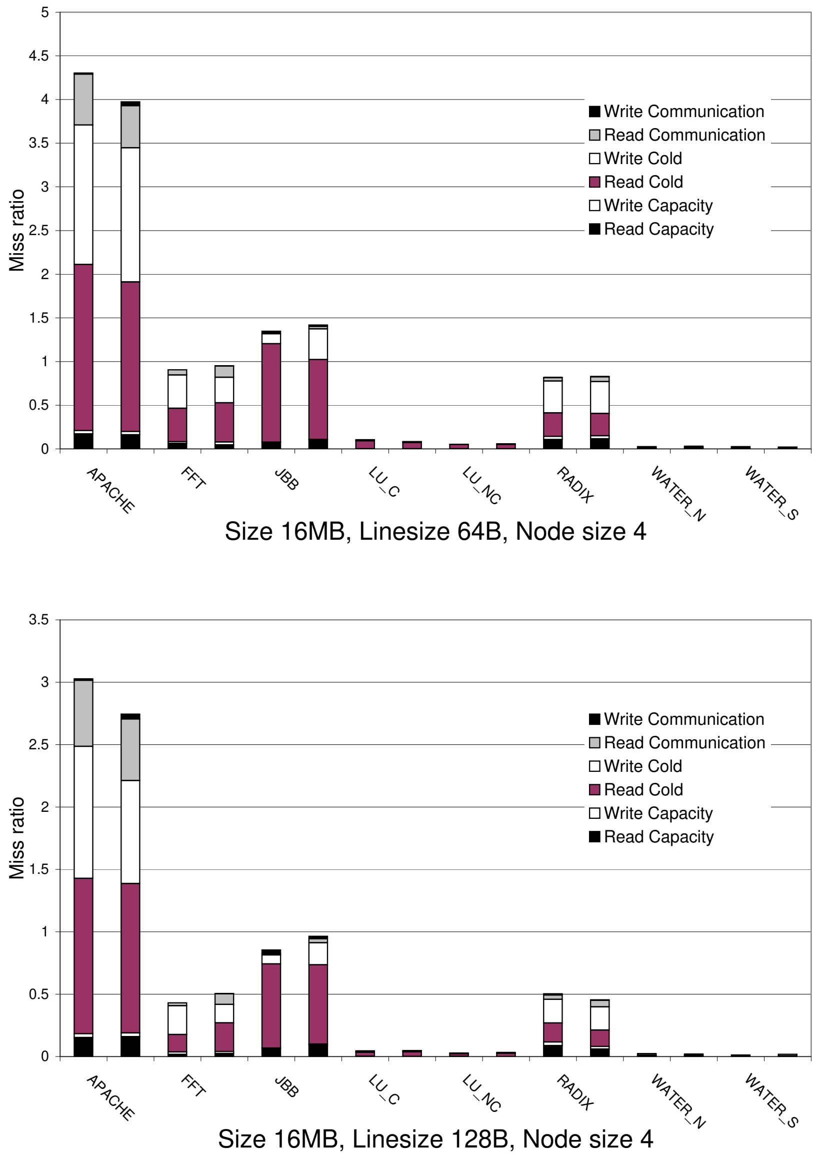
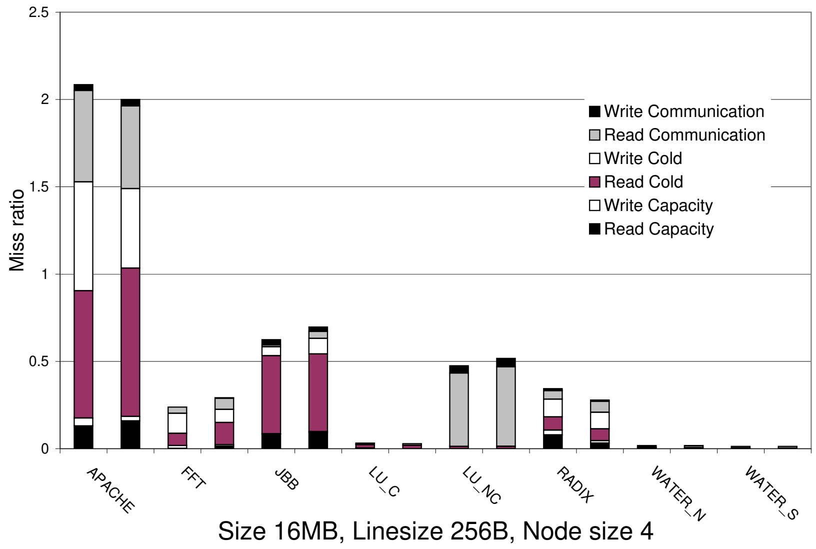


Figure 8: This diagrams show the miss ratio breakdown for the benchmarks APACHE, FFT, JBB, LU_C, LU_NC, RADIX, WATER_N and WATER_S. The cache size is 16M bytes and the cache-line size is 64 and 128 bytes.



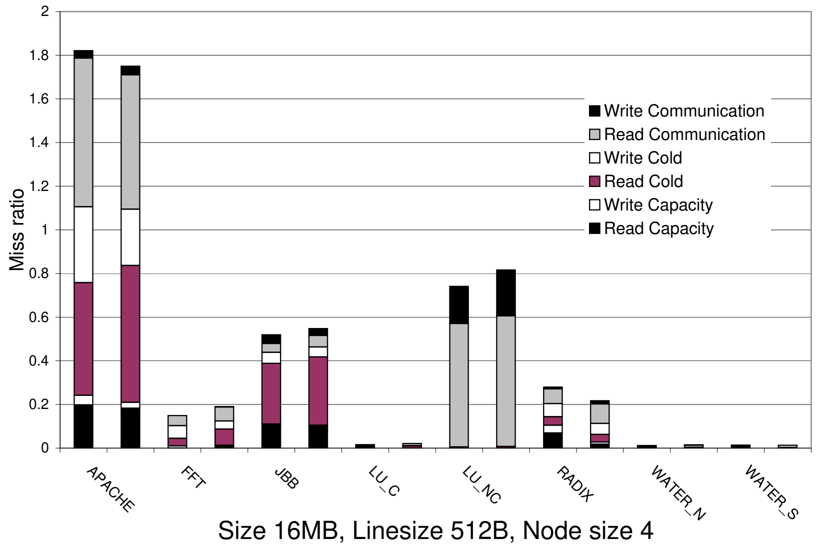
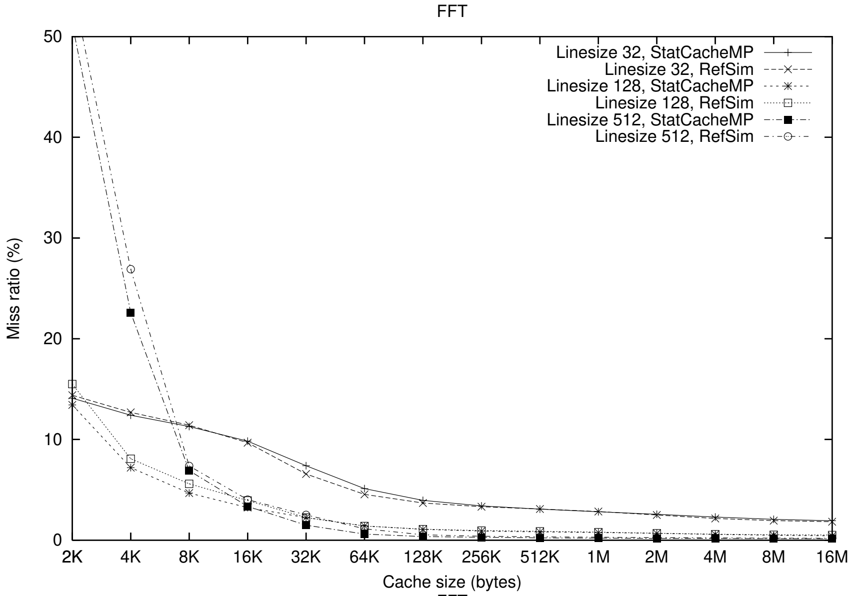


Figure 9: This diagrams show the miss ratio breakdown for the benchmarks APACHE, FFT, JBB, LU_C, LU_NC, RADIX, WATER_N and WATER_S. The cache size is 16M bytes and the cache-line size is 256 to 512 bytes.

Appendix B: Miss Ratio vs. Cache Size



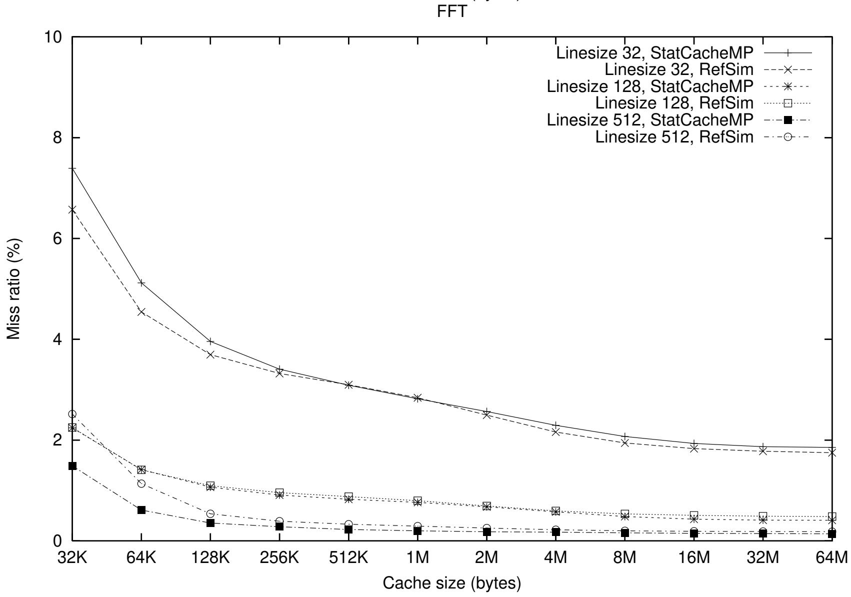
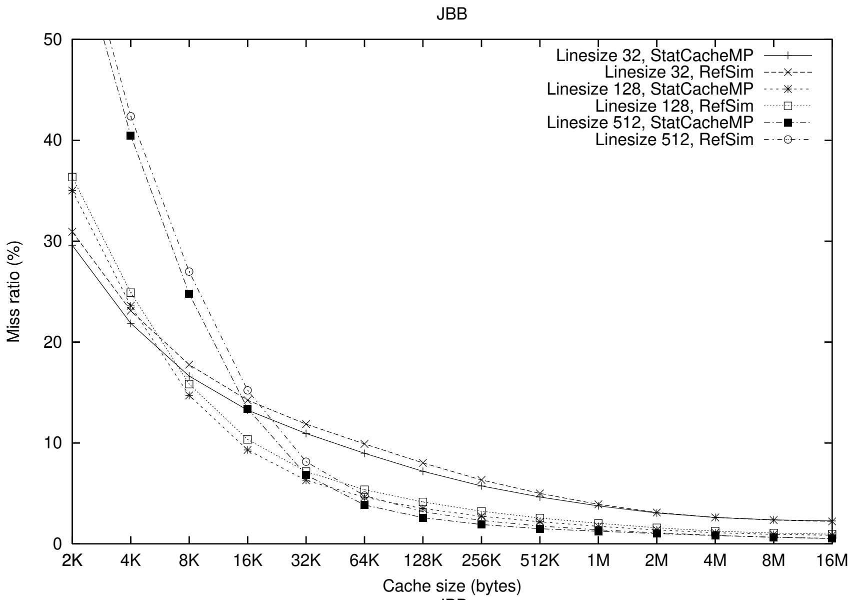


Figure 10: The graphs show the total miss ratio as a function of cache size for the benchmark FFT in two different scales. The cache-line sizes are 32, 128 and 512 bytes.



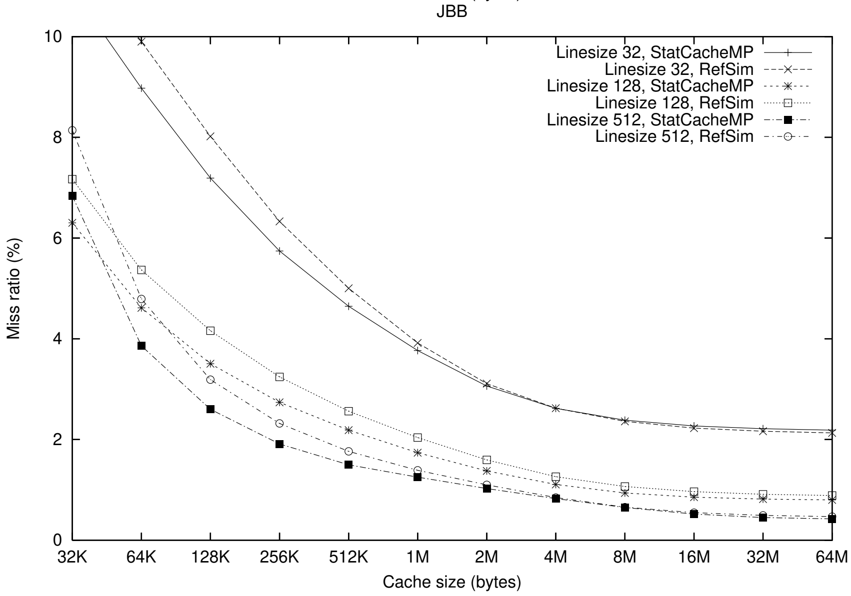


Figure 11: The graphs show the total miss ratio as a function of cache size for the benchmark JBB in two different scales. The cache-line sizes are 32, 128 and 512 bytes.

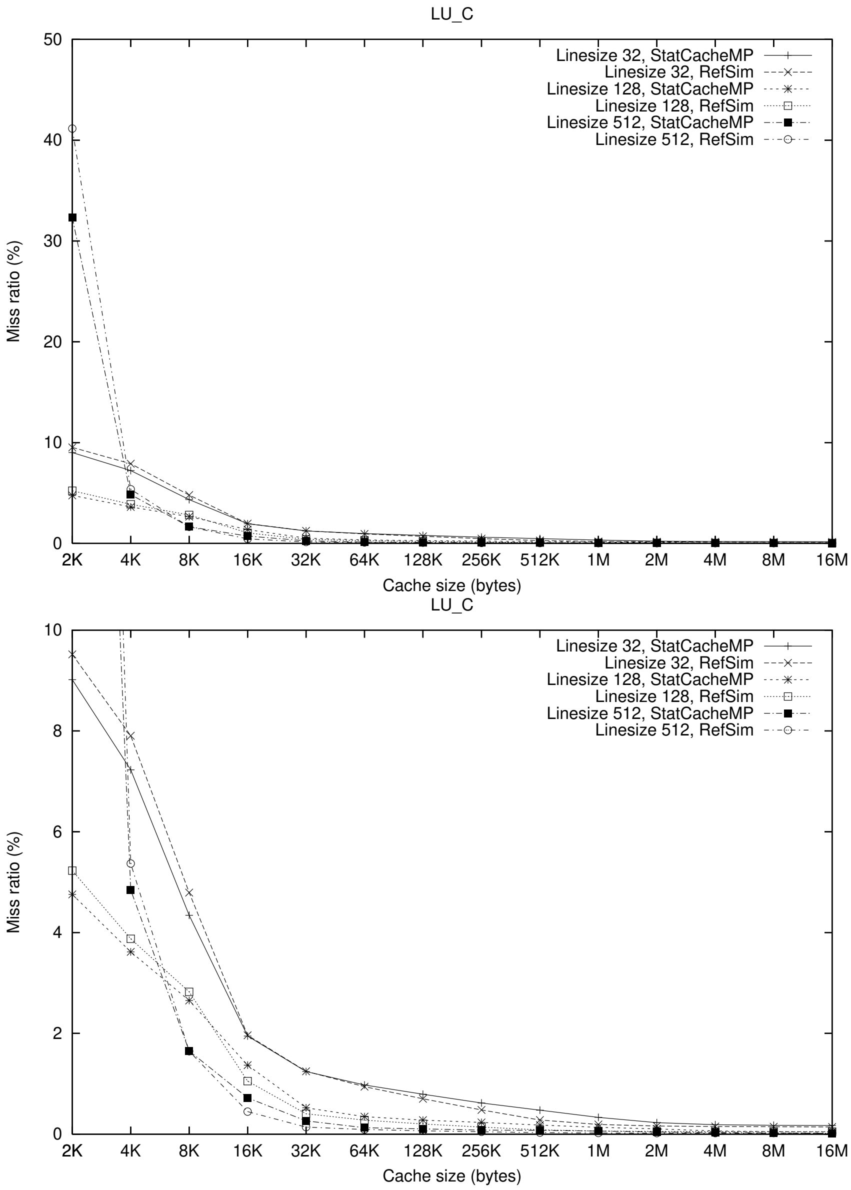
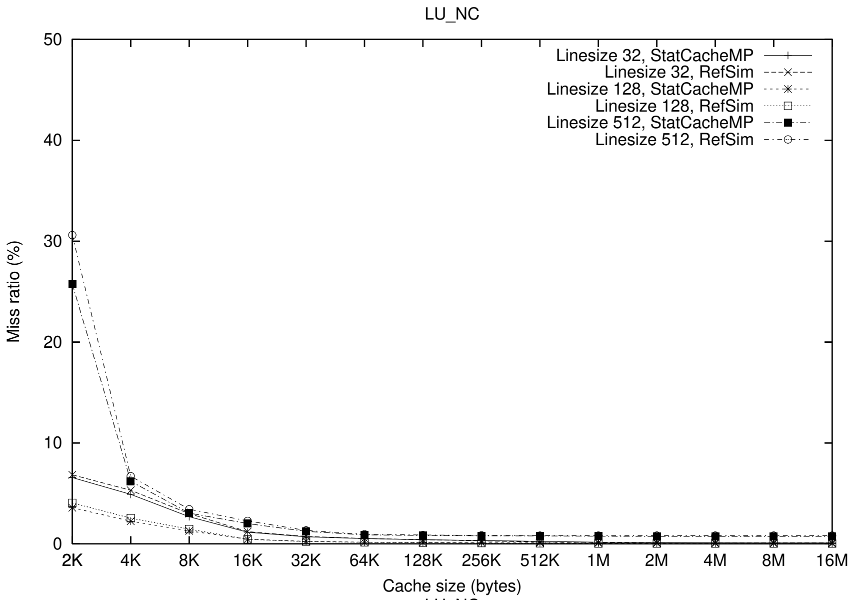


Figure 12: The graphs show the total miss ratio as a function of cache size for the benchmark LU_C in two scales. The cache-line sizes are 32, 128 and 512 bytes.



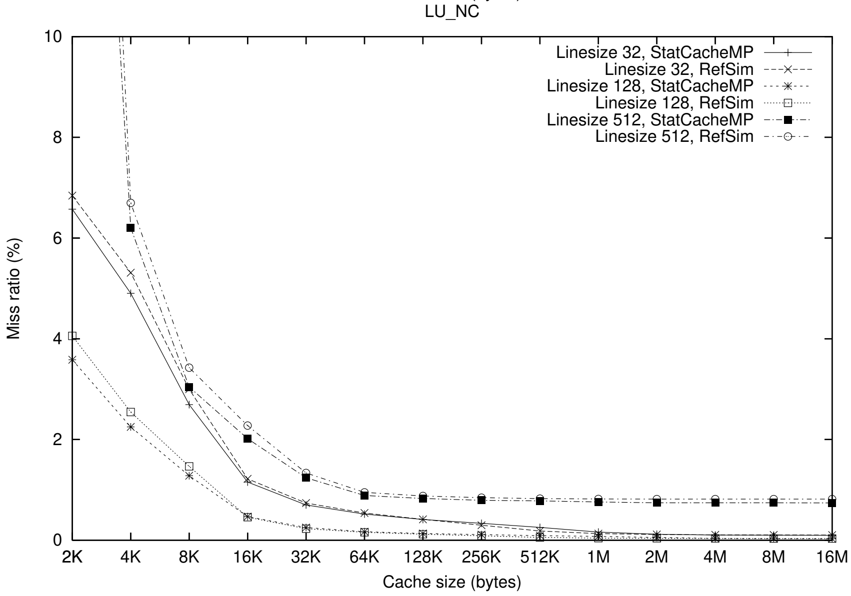
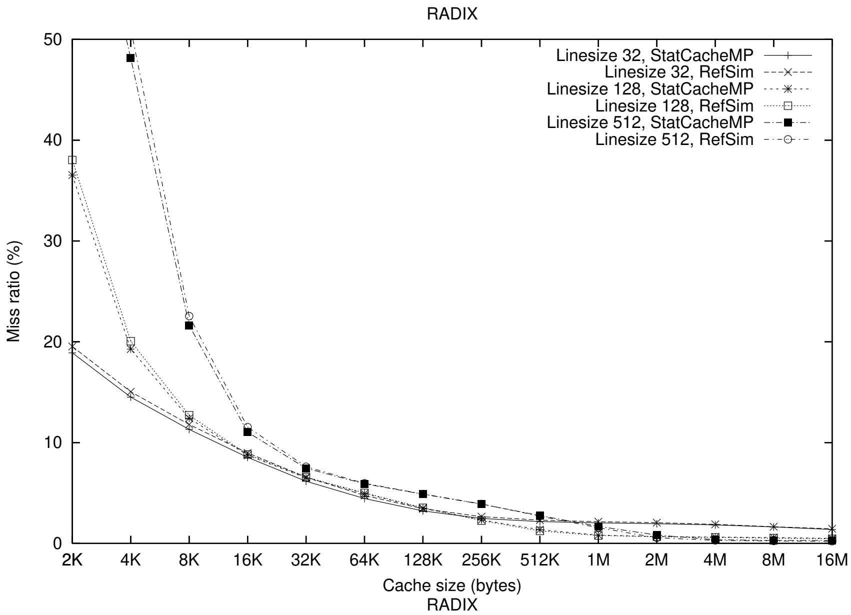


Figure 13: The graphs show the total miss ratio as a function of cache size for the benchmark LU_NC in two different scales. The cache-line sizes are 32, 128 and 512 bytes.



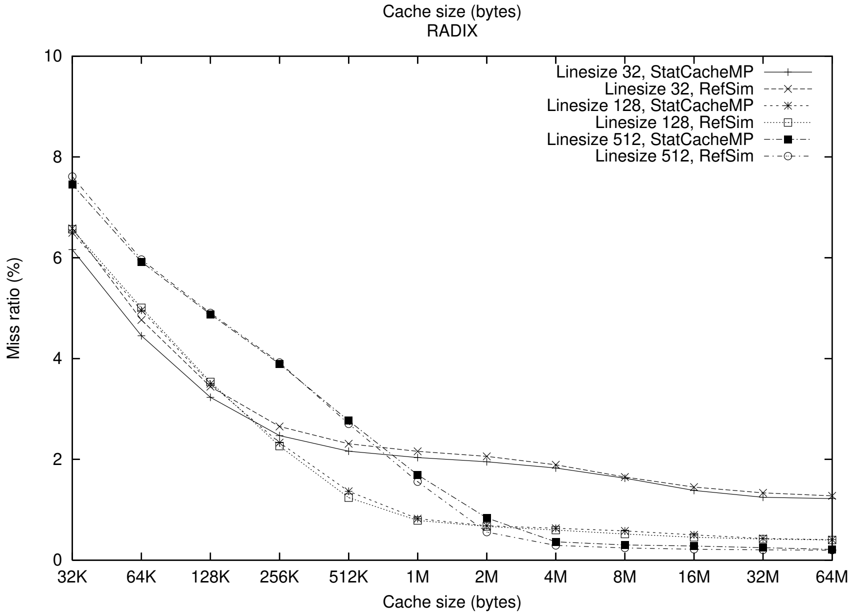


Figure 14: The graphs show the total miss ratio as a function of cache size for the benchmark RADIX in two different scales. The cache-line sizes are 32, 128 and 512 bytes.

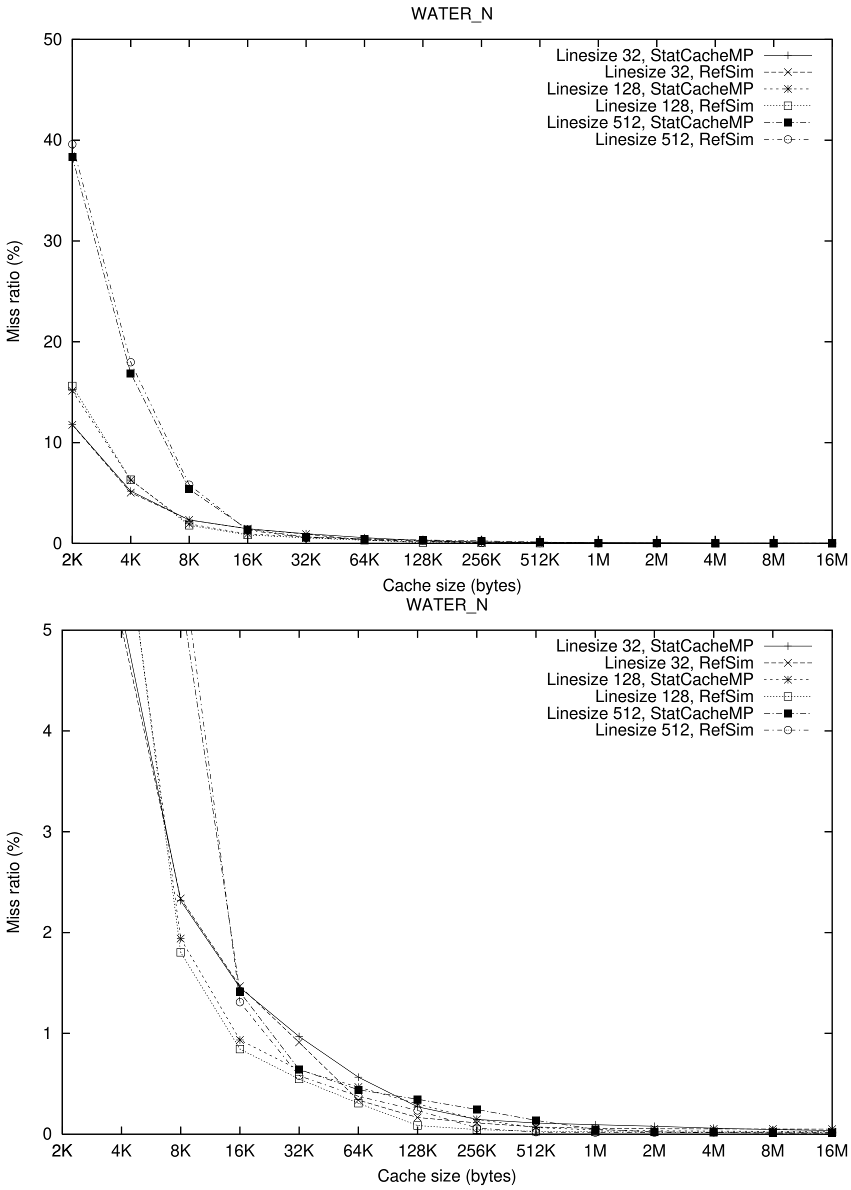


Figure 15: The graphs show the total miss ratio as a function of cache size for the benchmark WATER_N in two different scales. The cache-line sizes are 32, 128 and 512 bytes.

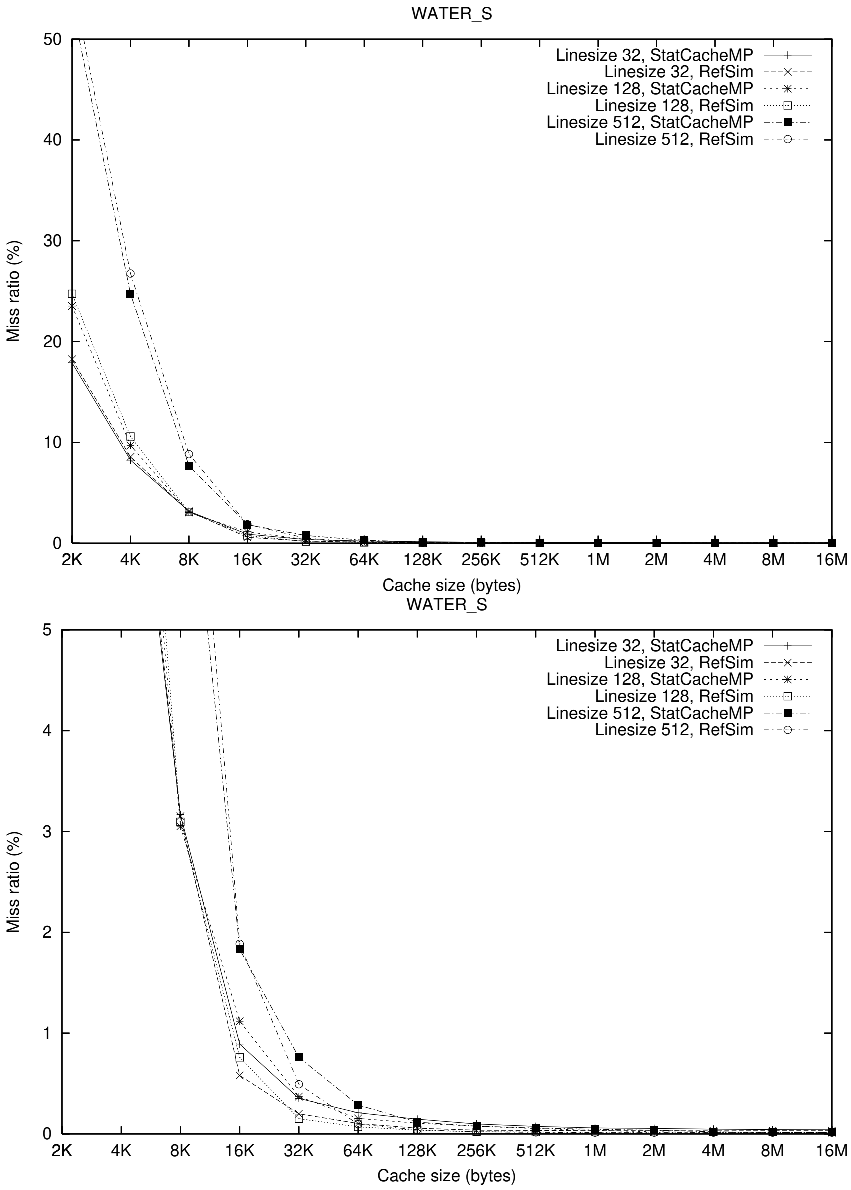


Figure 16: The graphs show the total miss ratio as a function of cache size for the benchmark WATER_S in two different scales. The cache-line sizes are 32, 128 and 512 bytes.

References

[1] E. Berg and E. Hagersten. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *Proceedings of International Symposium on Performance Analysis of Systems And Software*, Austin, Texas, USA, 2004.

[2] E. Berg and E. Hagersten. Fast Data-Locality Profiling of Native Execution. In *Proceedings of the International Conference on Measurement an Modeling of Computer Systems*, Banff, Canada, June 2005.

[3] K. Beyls, E. D'Hollander, and F. Vandeputte. Rdvis: A tool that visualizes the causes of low locality and hints program optimizations. In *Proceedings of the Fifth International Conference on Computational Science*, pages 166–173, Atlanta, USA, May 2005.

[4] K. Beyls, E. D'Hollander, and Y. Yu. Visualization enables the programmer to reduce cache misses. In *Proceedings of Conference on Parallel and Distributed Computing and Systems*, 2002.

[5] C. Cascaval and D. A. Padua. Estimating cache misses and locality using stack distances. In *Proceedings of International Conference on Supercomputing*, 2003.

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

[7] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

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

[9] S. Laha, J.A. Patel, and R.K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on computers*, 1988.

[10] P. Magnusson, F. Larsson, A. Moestedt, B. Werner, F. Dahlgren, M. Karlsson, F. Lundholm, J. Nilsson, P. Stenström, and H. Grahn. SimICS/sun4m: A virtual workstation. In *Proceedings of the Usenix Annual Technical Conference*, pages 119–130, 1998.

[11] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 2–13, New York, June 2004.

[12] 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.

[13] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for accurate and efficient simulation. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 2003.

[14] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. Using the SimOS machine simulator to study complex systems. *ACM Transactions on Modelling and Computer Simulation*, 7:78–103, 1997.

[15] X. Vera and J. Xue. Let's study whole-program cache behaviour analytically. In *Proceedings of 8th International Symposium on High-Performance Computer Architecture*, 2002.

[16] D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. *ACM SIGMETRICS Performance Evaluation Review*, 19(1), May 1991.

[17] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of International Symposium of Computer Architecture*, 2003.

[18] Y. Zhong, S. G. Dropsho, and C. Ding. Miss rate prediction across all program inputs. In *Proceedings of Parallel Architechtures and Compilation Techniques*, 2003.

Recent technical reports from the Department of Information Technology

2005-028 Erik Berg, Håkan Zeffer, and Erik Hagersten: *A Statistical Multiprocessor Cache Model*

2005-027 Timo Qvist: *Fast Simulation of Concurrent Agents with P-Nets - the GPSim tool*

2005-026 Erik Abenius, Fredrik Edelvik, and Christer Johansson: *Waveguide Truncation Using UPML in the Finite-Element Time-Domain Method*

2005-025 Paul Pettersson and Wang Yi (eds.): *Pre-Proceedings of the 3rd International Conference on Formal Modelling and Analysis of Timed Systems*

2005-024 Torsten Söderström, Erik K. Larsson, Kaushik Mahata, and Magnus Mossberg: *Approaches for Continuous-Time Modeling in Errors-in-Variables Identification*

2005-023 Per Lötstedt and Lars Ferm: *Dimensional Reduction of the Fokker-Planck Equation for Stochastic Chemical Reactions*

UPPSALA
UNIVERSITET

November 2005
ISSN 1404-3203
http://www.it.uu.se/