StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis

Erik Berg and Erik Hagersten

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

Technical report 2003-58
November 2003
ISSN 1404-3203

StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis

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

Abstract

The widening memory gap reduces performance of applications with poor data locality. This problem can be analyzed using working-set graphs. Current methods to generate such graphs include set sampling and time sampling, but cold start effects and unrepresentative set selection impair accuracy.

In this paper we present StatCache, a novel sample-based method that can perform data-locality analysis on realistic workloads. During the execution of an application, sparse discrete memory accesses are sampled, and their reuse distances are measured using a simple watchpoint mechanism. StatCache uses the information collected from a single run to accurately estimate miss ratios of fully-associative caches of arbitrary sizes and generates working-set graphs.

We evaluate StatCache using the SPEC CPU2000 benchmarks and show that StatCache gives accurate results with a sampling rate as low as 10^{-4}. We also provide a proof-of-concept implementation, and discuss potentially very fast implementation alternatives.

1 Introduction

Most modern high performance computers use cache memories to bridge the widening gap between DRAM access time and processor cycle time. This makes application execution time sensitive to cache behavior. To improve performance, it is often necessary to reduce the number of cache misses. To facilitate this, we need methods to measure cache behavior.

Cache misses can be divided into conflict, capacity and cold misses. These different types of cache misses have different causes and are subsequently reduced using different methods. For example conflict misses can often be reduced by padding, while blocking is effective to reduce capacity misses. This paper focuses on the often dominating capacity misses, and to reduce these misses we need to improve application data locality. An application with good data locality causes less cache misses because it often reuses recently accessed data and/or accesses data in the same cache line. To quantify the data locality, we can study the miss ratio of fully-associative caches and plot working-set graphs, that is, the miss ratio of a fully-associative cache plotted as a function of its size.

Simulating fully-associative caches is time consuming. For long-running applications the simulation time would be unbearable. Thus, some form of sampling is often used. The two common methods, time sampling and set sampling, both have their drawbacks. Time sampling requires very long warm up periods for large caches and set sampling may be very inaccurate.

We present StatCache, a novel probabilistic approach to estimate miss ratios of fully-associative caches for data-locality analysis. Rather than implementing a functional fully-associative cache simulator, StatCache uses a probabilistic model of a fully-associative cache to estimate its miss ratio. A major advantage of the described method is that it base its miss ratio estimates on information that is easy to collect by sparse sampling at run-time. In the paper we show that StatCache gives accurate results with a sampling as low as 10^{-4}. StatCache also overcomes the problem of cold-start effects. We present a proof-of-concept implementation with reasonable slowdown of the execution (5.8 on average), and suggest other potentially very fast implementations.

While this paper focuses on program optimization and tuning, the method could prove useful in other areas as well, such as workload characterization in computer architecture. An interesting possible application of StatCache is scheduling on clusters based on memory behavior.

2 Quantifying Data Locality

Before we go into the details of StatCache, we need to introduce a way to represent application data locality, show how it characterizes an application and how it can be approximated by sampling.

2.1 Reuse Distance

A running program reads and writes data in the memory. We number the memory addresses of the read and write operations and call them memory references. Let the reuse distance for a memory access to a cache-line-sized piece of data be the number of memory references since the the same piece of data was last accessed.

Memory reference counter	1	2	3	4	5	6	7	8	9	10
Referenced cache line	B	D	D	D	C	B	C	C	A	B
Data reuse distance	-	-	0	0	-	4	1	0	-	3

Figure 1: Example of the data reuse distance. B is accessed the first time by reference 1, and the second time by reference 6. This gives the reuse distance 6-1-1=4 for reference number 6. The reuse distance of reference 3 is 0 because it accesses the same cache line as the previous reference.

Figure 1 illustrates the concept of reuse distance. Cache line B is accessed in reference one, and the next time in reference six. Thus, the reuse distance for reference six is four. The reuse distance for the first reference to a cache line is undefined.

The reuse distance defined above counts all intermediate memory references, in contrast to the stack distance[21] that only counts the number of different references since the last reference to the data. For example, in Figure 1 the stack distance of the sixth reference is only 2.

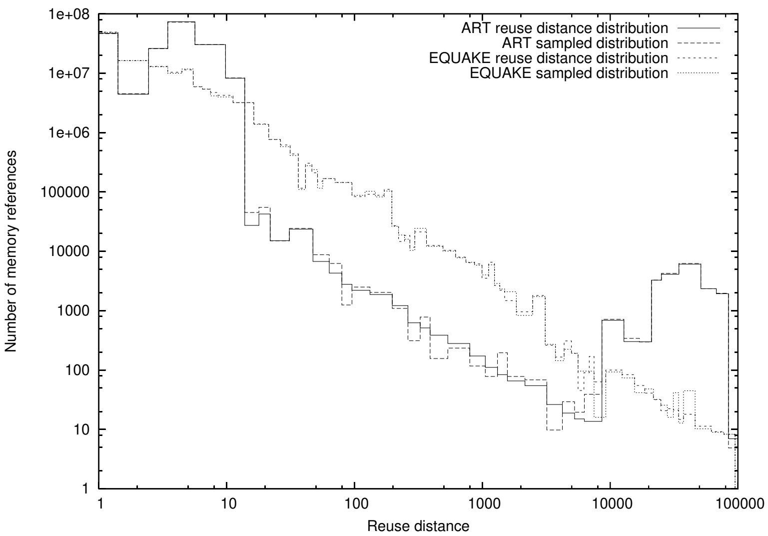


Figure 2: The diagram shows the reuse-distance distribution, i.e. the number of memory references as a function of reuse distance, for the SPEC benchmarks art and equake. It shows the distributions of complete memory access traces and of sampled traces. The sampled distributions are very similar to their respective unsampled distributions.

2.2 Sampling

The distribution of the reuse distances is like a finger print of application data locality. Figure 2 shows the reuse distance distributions for the SPEC benchmarks art and equake. The x-axis is the reuse distance and the y-axis is the number of memory references for a given reuse distance. Many references with low reuse distance indicate good data locality. The benchmark art have a bump for large reuse distances, which means that many memory access have poor temporal locality. This does, however, not automatically mean that the miss ratio is high. The cache may be able to keep data long enough to prevent the accesses with high reuse distances to result in cache misses.

As long as we are only interested in the shape of the distribution, it need not be based on every memory access, so StatCache uses sampling to estimate the distribution. Long continuous traces are not needed, but instead StatCache samples discrete memory references and measures their reuse distance. Figure 2 shows distributions based on complete memory access traces and distributions approximated with a sampling rate of 10^{-4}.

3 StatCache

A working-set graph is a plot of the miss ratio of a fully-associative cache as a function of its size[24]. The central idea of StatCache is to use sample-based estimates of reuse-distance distributions to generate working-set graphs. StatCache uses fully-associative caches with random replacement instead of LRU replacement, but as we will show later, the results are similar.

3.1 StatCache Theory

The probability that referenced data resides in the cache-memory can be described in terms of reuse distance. Assume that a cache is fully associative, has L cache lines and uses random replacement policy. The probability is $(1 - 1/L)$ that a cache line remains in the cache after one cache miss and $\left(1 - \frac{1}{L}\right)^n$ after n cache misses. Let $f(n)$ be the probability that a cache line *do not* remain in the cache after n cache misses, i.e.

$$f(n) = 1 - \left(1 - \frac{1}{L}\right)^n \tag{1}$$

Let $r(i)$ denote the probability that memory reference i causes a cache miss, i.e. $r(i)$ = P(reference i misses). Take two memory references i and j, where

$i \leq j$. We estimate the total number of cache misses between i and j with the sum $r(i+1) + r(i+2) + \cdots + r(j-2) + r(j-1) = \sum_{i+1}^{j-1} r(i)$

Denote the reuse distance of the i-th memory reference with $A(i)$. For example, Figure 1 gives $A(6) = 4$ and $A(7) = 1$. According to the definition of the reuse distance, reference i accesses the same cache line as reference $i - A(i) - 1$. Thus, until reference i, from the reference that last accessed the same cache line, the number of cache misses is $\sum_{k=i-A(i)}^{i-1} r(k)$

Combining this with equation 1 yields the probability that the i-th reference will miss.

$$r(i) = f\left(\sum_{k=i-A(i)}^{i-1} r(k)\right)$$

Summing over every memory reference yields the total number of misses of the application:

$$Totalmisses = \sum_{i=1}^{N} f\left(\sum_{k=i-A(i)}^{i-1} r(k)\right) \tag{2}$$

where N denotes the total number of memory references.

But the sum over all $r(i)$ is also equal to the total number of misses,

$$Totalmisses = \sum_{k=1}^{N} r(k) \tag{3}$$

Putting equations 2 and 3 together gives:

$$\sum_{i=1}^{N} f\left(\sum_{k=i-A(i)}^{i-1} r(k)\right) = \sum_{i=1}^{N} r(i) \tag{4}$$

Hence, we have a relationship between the reuse distances, $A(i)$, and miss probabilities, $r(i)$.

Assume that the miss probability function $r(i)$ can be approximated with a constant R. Then equation 4 becomes

$$\sum_{n=1}^{N} f\left(A(i) \cdot R\right) = \sum_{i=1}^{N} R = R \cdot N$$

We can now use the sampled reuse distance distribution. Let h be the distribution. $h(0)$ is the number of references with reuse distance 0, $h(1)$ is the number of references with reuse distance 1, etc. There are $h(i)$ j's for which $A(j) = i$. Thus[1]:

$$\sum_{i=1}^{\max_{0 \leq i \leq N}(A(i))} h(i) f(i \cdot R) = R \cdot N \tag{5}$$

This equation can be solved numerically for any given distribution h.

3.2 Varying Miss Ratios

The assumption that the miss ratio is constant is not valid for all applications. Our early experiments showed that the method above gives accurate results for applications with constant miss ratio, but gives unacceptable errors when the miss ratio varies during execution. The solution is to split the execution into small time slots and estimate the miss ratio of each time slot with equation 5, using a separate reuse distance distribution for each time slot. The arithmetic mean of the miss ratios of the time slots approximates well the miss ratio of the application.

3.3 Generating Working-Set Graphs

Remember that the number of cache lines, L, can be set arbitrarily before solving equation 5. Thus, a single run generates all information that is needed to calculate miss ratios of fully associative caches of arbitrary sizes. By solving equation 5 for different values of L, working set graphs can easily be generated. L need not be a power of 2, why smoother curves can be generated.

3.4 Cold Misses

StatCache will simply ignore all references to memory that have not been accessed before, because the reuse distance is not defined for the first references to a piece of data. Therefore, the miss ratios StatCache gives will not include cold misses. The good thing is that capacity misses are separated from both conflict misses and capacity misses. However, the number of cold misses is usually small, so the effect can often be ignored.

3.5 Error Estimate

If the error is unacceptable, the sampling rate should be increased. Optimization decisions are often based on comparative experiments, and the user, or the compiler

[1]Note that we can start the left hand side sum from 1 because $h(0)f(0 \cdot R) = 0$.

in an automated compiler-feedback system, must know the validity of the results. This is important also for architectural studies [1].

We use the statistical method bootstrapping [5] to estimate the sampling error. By repeated resamples we can estimate the expected spread. Assume that the execution is divided into n time slots, as described in Section 3.2. First, calculate the miss ratio for every time slot. Denote the set of results S. Then resample n values from S with replacement and calculate the mean miss ratio. Repeated resampling gives a distribution of the miss ratio without performing the experiment several times. According to the bootstrap theory, the width of the distribution excluding the upper and lower 2.5 percentiles, gives a 95 percent confidence interval. The error estimate is displayed as error bars in the working set graph.

4 Experimental Evaluation

To evaluate the accuracy of StatCache, we compare working-set graphs generated by StatCache to corresponding graphs generated by fully associative caches. We simulate both StatCache and the fully associative caches to evaluate StatCache as such, and exclude possible deficiencies in an implementation.

4.1 Experiment Setup

We used trace driven simulation in our experiments. The Simics [18] full system simulator, that simulates an Sun UltraSPARC II workstation-like computer, was used to generate the traces. To cut simulation time, we used the twenty benchmarks from the SPEC CPU2000 suit that are available with large reduced input data sets [13]. To simulate fully associative caches of various sizes for every SPEC benchmark using the reference inputs would be too time consuming. The length of the reduced traces are between $90 \cdot 10^6$ and $800 \cdot 10^6$ references.

The StatCache simulator sequentially reads the trace and increments a common memory reference counter for every memory reference. Using a random generator it samples memory references and adds their addresses and the corresponding value of the memory reference counter to a list of watchpoints. When an address from the trace references the same cache line as a watchpoints on the list, the reuse distance is added to a reuse distance distribution, like that in Figure 2, and the watchpoint is removed. The execution is split into time slots as described in section 3.2 to get accurate miss ratio estimates. Miss ratios are calculated with equation 5.

We have also simulated fully associative caches with random replacement from 2 K Byte to 4M Byte. The miss ratios StatCache computes exclude cold misses, and to examine the effect of this, the results come in two flavors, with (FA_RND) or without cold misses (FA_RND_NOCOLD). If the cold misses are negligible we only present miss ratios including cold misses, otherwise we present both results. Fully associative caches with LRU replacement (FA_LRU) have also been simulated to investigate if there is a significant difference between the LRU and random replacement policies. The cache line size is 32 bytes throughout the paper.

4.2 StatCache Accuracy

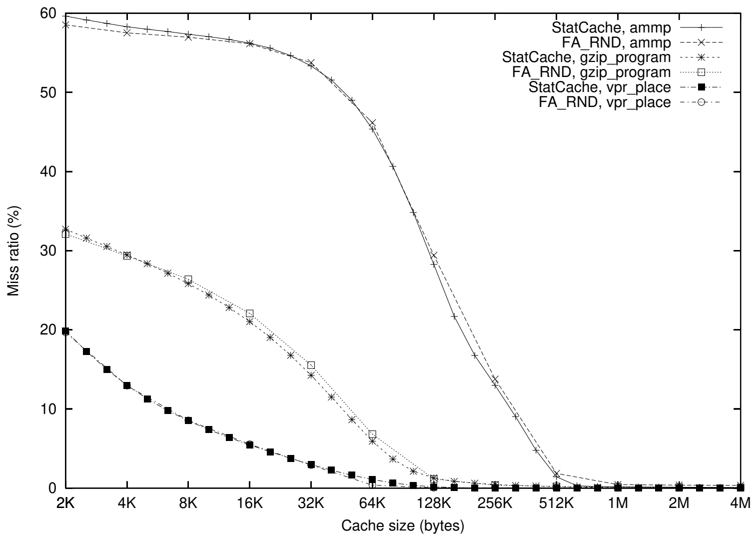
Figures 3 shows miss ratios of fully associative caches with random replacement (FA_RND) compared to miss ratios estimated by StatCache using a sampling rate of 10^{-4}. StatCache manages to accurately estimate the absolute values of the miss ratios, to capture the overall shape of the working set graphs well, and to give stable, smooth curves. In the case of art, twolf, and vpr_route, they are almost identical. For perlbmk, the curves deviate between 2K Byte and 4K Byte because we have only simulated fully associative caches with power of two sizes. Note that the low sampling ratio gives only 9000 - 80000 samples per application, which is less than the warm up period for large caches. Equake, mcf and mesa have a large fraction of cold misses. For these applications we also show graphs for fully associative caches with the cold misses removed (FA_RND_NOCOLD).

4.3 Impact of Replacement Policy

Using a random or LRU-replacement policy have little effect on the miss ratio for most applications, but small differences exist, as Figure 6 shows. For bzip2_graphic, LRU-replacement gives a smaller miss ratio than random-replacement for small caches, but the difference vanishes as the cache size grows.

For art, LRU-replacement policy tend to give larger miss ratios than random-replacement policy for caches between 256K byte and 1M byte. This is typical of applications that traverses large amounts of data without reusing it for a long time. Compare with Figure 2, where the reuse-distance distribution of art has a bump for large reuse distances. For some applications, like gzip_program, the difference is very small. Mcf is the only benchmark where the replacement scheme affects the overall shape of the graph, but the differences is still limited.

Another reason that random replacement can be used instead of LRU replacement to generate working-set graphs is the following. Consider a direct mapped cache. If the access pattern of an application causes all memory references to hash uniformly across the cache lines, the cache no longer suffers from its lack of associativity. When a cache miss occurs, each cache line is evicted by the same probability. A fully-associative cache with random-replacement also selects victims uniformly across all cache lines, and must therefore have the same miss ratio



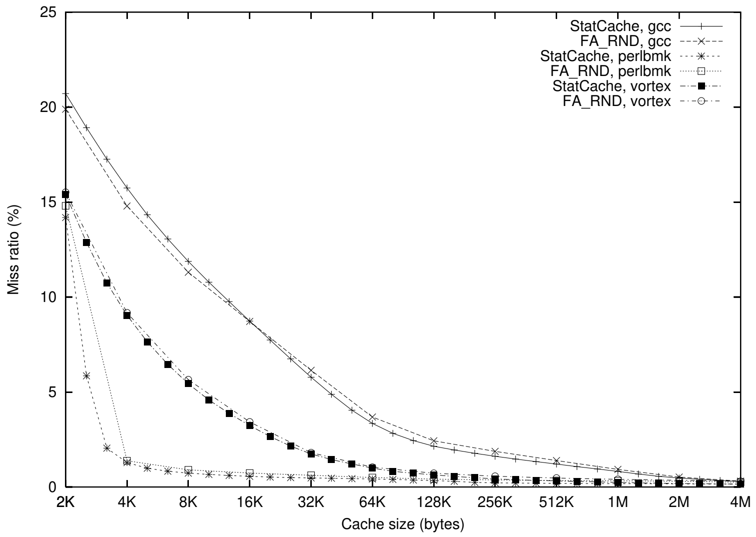


Figure 3: StatCache compared to fully-associative caches with random replacement (labeled FA_RND). The graphs show the miss ratio as a function of cache size for 20 benchmarks. StatCache accurately estimates the miss ratio of the fully-associative cache and the general behavior is well captured. The graphs for equake and mcf show that StatCache does not measure cold misses. StatCache is much closer to the graphs without cold misses (FA_RND_NOCOLD) than the graphs that includes cold misses (FA_RND). The figure is continued on next page.

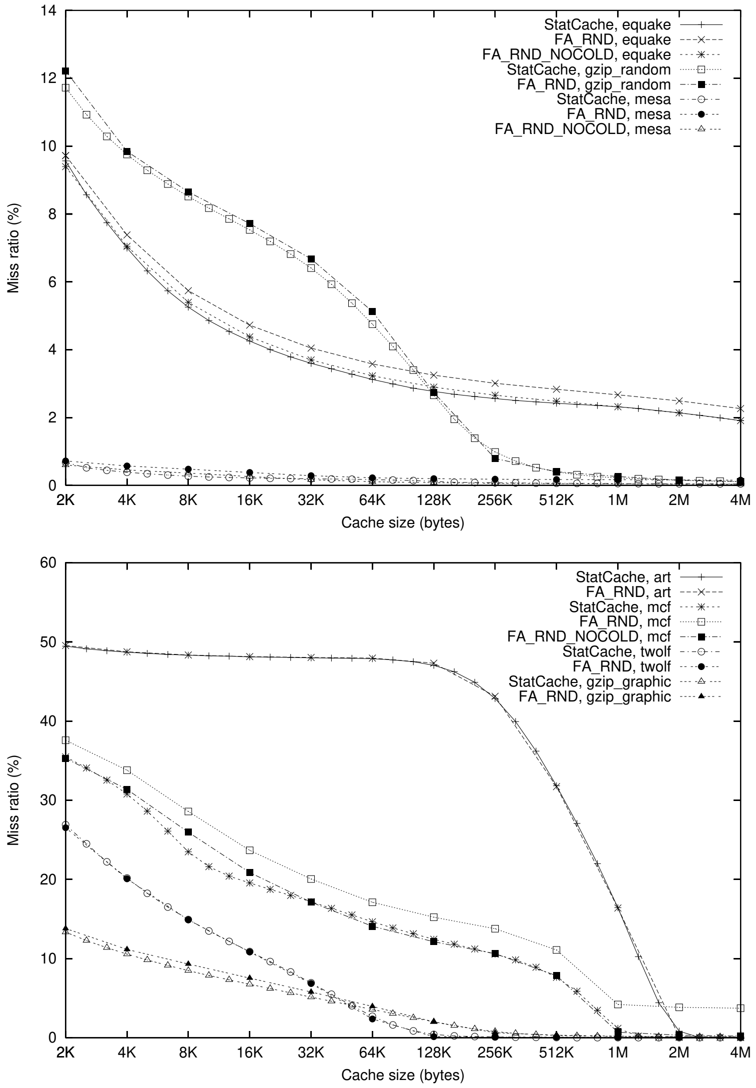


Figure 4: StatCache compared to fully-associative caches with random replacement (labeled FA_RND). The figure is a continuation from previous page.

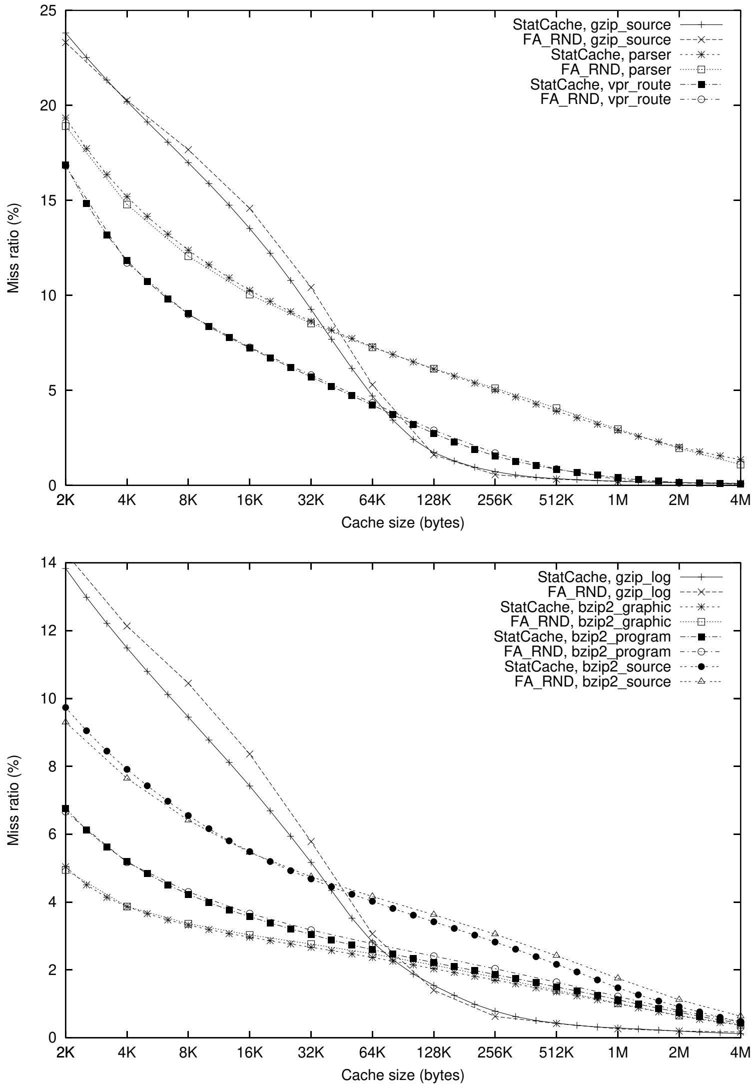


Figure 5: StatCache compared to fully-associative caches with random replacement (labeled FA_RND). The figure is a continuation from previous page.

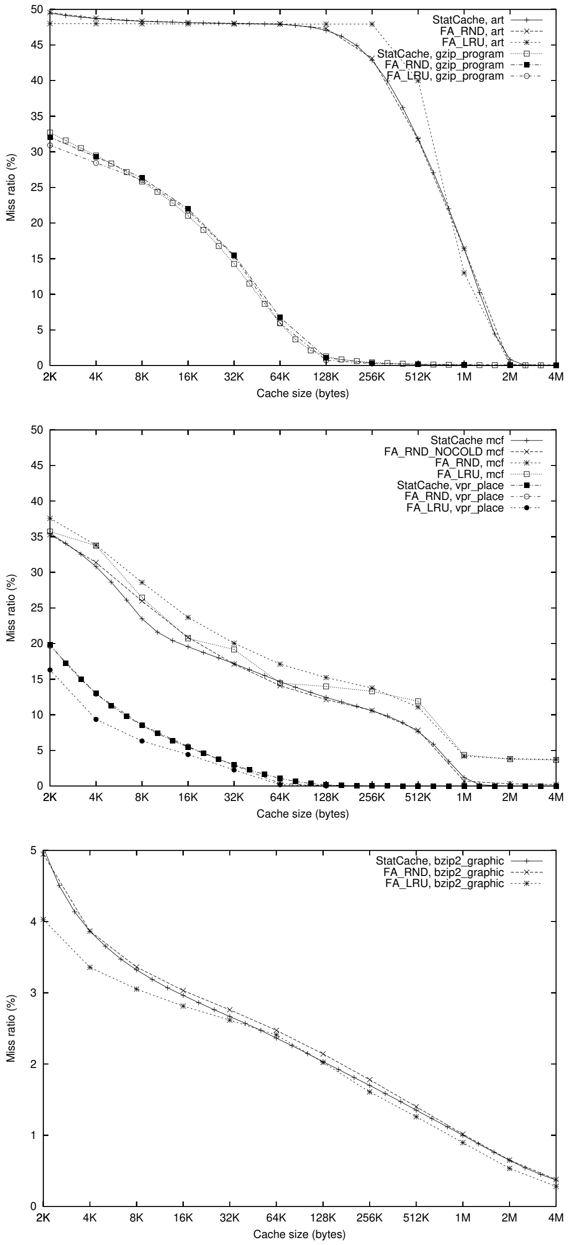


Figure 6: Examples of the difference between LRU-(FA_LRU) and random-(FA_RND) replacement policies of fully-associative caches on the benchmarks art, gzip_program, mcf, vpr_place, and bzip2_graphic. The graphs show that the two replacement policies usually give similar miss ratios.

as the direct-mapped cache in the absence of conflict misses. A similar argument holds for set-associative caches with random replacement.

4.4 Accuracy of the Error Estimate

Examples of the error estimate discussed in section 3.5 are shown in Figure 7. The error bars indicate the confidence interval.

Figure 7 shows what happens to the error estimate and accuracy when the sampling rate is changed, a higher sampling rate gives better accuracy and tighter error bounds. The error bars for mcf blows up for sampling rate 0.00002, indicating that the sampling rate should be increased.

Unfortunately, the error estimate is not absolutely accurate. It only takes sampling variations into account and not approximations in the method as such. Experiments showed that the error interval covers the correct value in 90 percent of the cases, but from the bootstrapping method we'd expect a 95 percent confidence interval. Despite this, the error estimate provides guidance for setting the sampling rate.

4.5 StatCache vs. Set Sampling

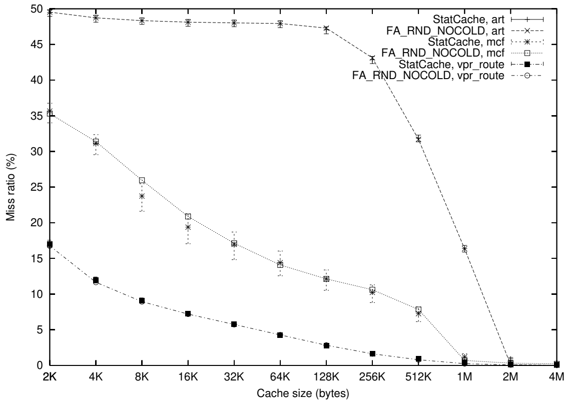
Set sampling using highly-associative caches is an alternative for studying data locality, but the accuracy is a problem. Figure 8 shows set sampling compared to simulated set-associative caches. The solid line shows the miss ratios of simulated 8-way caches, and the points show set sampling estimates using every sixteenth set. The figure illustrates the large spread of the results that set sampling suffers from.

Figure 9 shows a comparison of the accuracy of set sampling versus StatCache. The staples show the root mean squared error. StatCache has lower error than set sampling for all benchmarks and StatCache also cuts the peak errors. Set sampling is up to 7 percent off, while StatCache keeps the error below 0.9 percent. The sampled sets could be changed during execution to improve accuracy, but the new sets must be warmed up. Large caches requires long impractical warm-up periods.

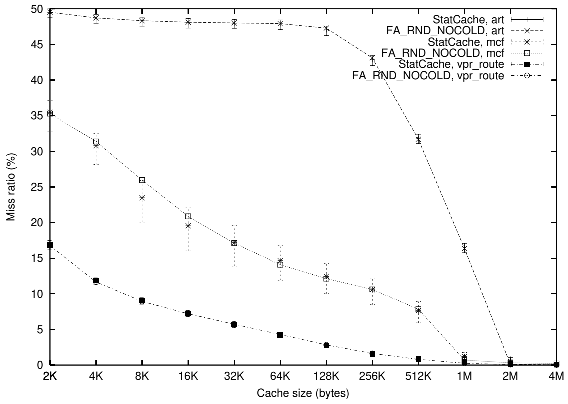
5 Implementation

A StatCache implementation must provide a sampling mechanism, a watchpoint mechanism, and a memory reference counter. The sampling mechanism randomly selects memory access instructions and sets watchpoints on the cache lines they access. The watchpoint mechanism detects cache line reuse, and the memory ref-

Sampling rate: $2 \cdot 10^{-4}$



Sampling rate: $1 \cdot 10^{-4}$



Sampling rate: $2 \cdot 10^{-5}$

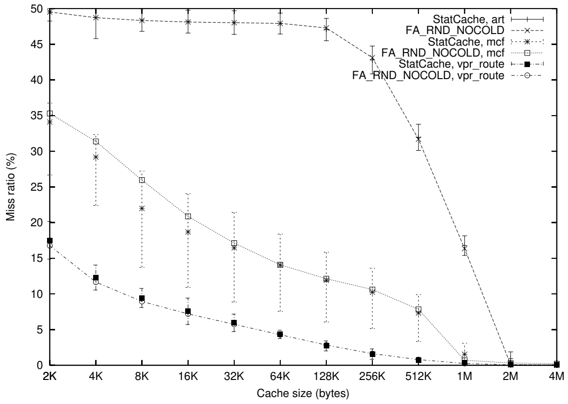


Figure 7: Example of error estimates for different sampling ratios. Note how the error bars grow as the sampling rate decreases. The error bars for mcf are very large for 0.00002 sampling rate, indicating that the sampling ratio is too low.

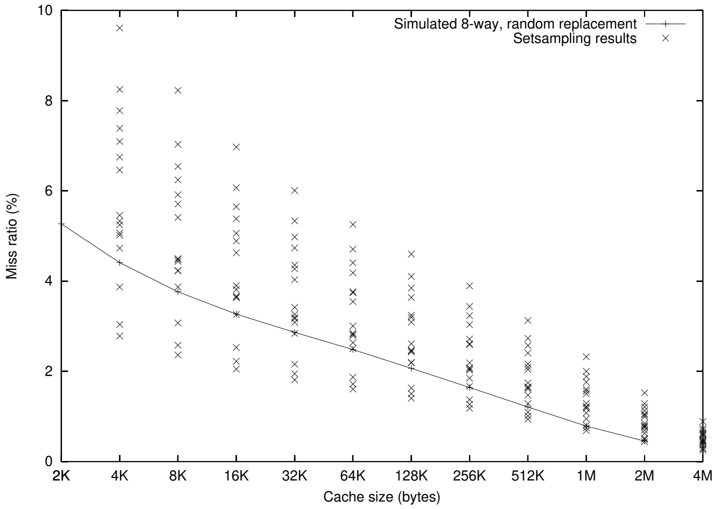


Figure 8: An example, bzip2_program, illustrating the large spread of estimated miss ratios when using set sampling. Crosses are different values from set sampling. This large spread is actually typical of set sampling and much reduces its use in comparative experiments.

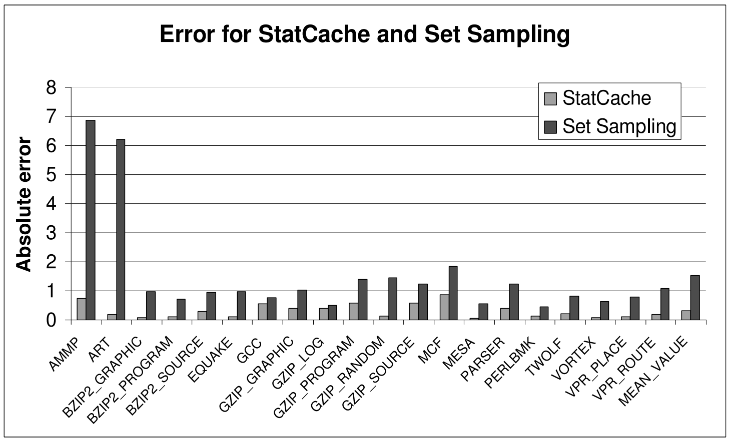


Figure 9: Deviation from the correct miss ratio for StatCache and set sampling. In the experiment, the sample rate for StatCache was set to 10^{-4} and every sixteenth set was used for the set sampling. The deviation of StatCache never exceeds 0.9 percent, while set sampling deviates almost 7 percent in the worst case.

erence counter is used to determine reuse distance when the watchpoint mechanism has detected reuse of a cache line.

StatCache only need very sparse samples, so the run time overhead introduced by the sampling mechanism can be limited. The performance will therefore largely be determined by the watchpoint mechanism that must check every memory access and determine if it hits a watchpoint set by the sampling mechanism. The overhead of StatCache would be very small if this check could be performed without extra work for the processor.

Most modern CPUs have hardware memory reference counters that can be used to select samples using overflow traps. Timers could also be used in a similar manner. Code instrumentation is an alternative we use in the proof-of-concept implementation.

Some architectures provide a limited number of hardware watchpoints and the memory management and the TLB may be used as a course grain watchpoint mechanism, but must be complemented by a fine grain filter. Most modern computers use ECC protected memory. If the processor can set the correction bits and trap precisely on error, the ECC-bits could be used as an efficient watchpoint mechanism. In our proof-of-concept implementation we use an efficient hash table lookup as watchpoint mechanism.

5.1 Proof-of-Concept Implementation

The proof-of-concept implementation is based on code instrumentation. We use the SAIT[10] SPARC assembly code instrumentation tool to insert a small piece of code, called snippet, next to every load and store instruction. The application is first compiled into assembly code using the usual C or F90 compiler, then instrumented, and finally compiled into object files that are linked with the StatCache runtime system.

The inline code snippet first decrements the reference counter. If the reference counter reaches zero it calls a sample handler that sets a watchpoint on the accessed cache line and resets the reference counter with a random value that indicates when the next sample should be taken. The snippet also uses a hash table to check for watchpoints and, if it's a hit, calls a watchpoint handler. The watchpoint handler removes the watchpoint, calculates and records the reuse distance, and updates the hash table.

The the proof-of-concept implementation based on code instrumentation is very similar to the simulated StatCache described in Section 4.1, because the code instrumentation makes it very deterministic. If we had used the instrumentation tool to generate the traces we used for the evaluation, the proof-of-concept implementation would behave exactly as the simulated StatCache.

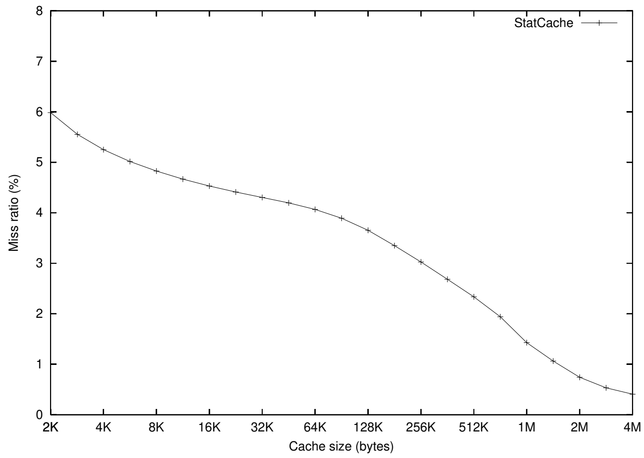


Figure 10: The output from the proof-of-concept implementation running 188.ammp with reference input. Observe the different shape compared to the corresponding curve in Figure 3 where we used a reduced input.

5.2 Performance of the Proof-of-Concept Implementation

We used the SPEC reference inputs to evaluate the performance of the proof-of-concept implementation on ten benchmarks [2]. We run the benchmarks on a Sun E450 with four 450MHz Ultra SPARC II CPUs. The average slowdown was 5.8 times on optimized code[3] and varied between 2.8 (187.facerec) and 8.3 (173.applu). The slowdown is likely to depend on instrumentation technique and post optimizations. The proof-of-concept implementation instruments optimized code, but do not apply any post instrumentation optimization, which would probably lower the slowdown considerably.

Figure 10 shows the working-set graph of 188.ammp with the the proof-of-concept implementation and reference input. Observe that the shape of the curve and the absolute values deviate from the results presented in Figure 3. This emphasizes the importance of performing measurements on realistic data sets.

[2]168.wupwise, 171.swim, 172.mgrid, 173.applu, 177.mesa, 178.galgel, 179.art, 183.equake, 187.facerec, and 301.apsi

[3]The uninstrumented code we used as baseline were compiled with Sun Forte C and Fortran compiler, version 5.4 and 7 respectively. Compiler flags for C/Fortran: -xO5 -xarch=v8plus. Same or lower optimization were used for instrumented codes.

5.3 Limitations of the Instrumentation Tool

The implementation has a few limitations. It can only instrument code compiled from source, cannot handle for example inline math functions, and allocates three of the thirty-two integer registers on the SPARC processor for use in the snippets. The instrumentation procedure, compilation, instrumentation and recompilation, is also lengthy. However, these are limitations of the proof-of-concept implementation, and not of StatCache as such. For example ATOM [8] and EEL [14] can instrument binaries, and the code instrumentation could be integrated in a compiler. It is outside the scope of this paper to show this.

6 Related Work

There are several ways to investigate an applications memory and cache behavior. Depending on the detail necessary, one can choose between detailed full system computer simulation, instruction set simulation, user level simulation or code instrumentation, source code instrumentation, trap driven cache simulation or measurements on hardware using hardware profiling support.

Full system simulators include Simics [18] and SimOS [16]. They allow very detailed cache simulations, but suffer from large slowdown and therefore are of limited use for studying very large workloads. Many tools have been built upon binary code instrumentation tools like DIOTA [17], ATOM [8] or EEL [14]. Examples are SIGMA [7], CPROF [15] and MemSpy [19][20], which are much faster than simulators but their slowdowns are still considerable, between 5 and 50 times is common. They can simulate caches to the desired detail, but cannot capture operating system interaction. Source instrumentation have also been explored, for example in MHSIM [11].

Sampling reference traces is used to speed up cache hierarchy simulation. It can be applied to all levels of detail of computer and application simulation. The common methods to sample are time sampling and set sampling. In time sampling a cache model simulates continuous sub-traces from the complete memory reference trace. This is explored in papers [25][12] [6][9]. It works well for smaller caches, but the need for long warmup periods makes time sampling less suitable for large caches. Set sampling is another approach, were only a fraction of the sets in a set associative cache is simulated [12] [6]. It generally suffers from poor accuracy and can only be used as a rough estimate. Trap driven trace generation has also been suggested [23]. It can trace unmodified code, but requires OS modification.

Hardware monitoring tools collect statistics from hardware and present the information in aggregated form to the user. Examples are DCPI [2], which uses an advanced hardware support to collect detailed information to the programmer, and

PAPI [3] which is a common programming interface to access hardware monitoring aids. Histogramming and tracing hardware may be used to detect for example cache conflicts [22] and locate problem areas [4]. The execution time overhead is very small, but they can only provide information about the configuration of the current machine. Interference between different applications may also be a problem. Accurate measurements require one program at a time running on the system.

7 Conclusions

In this paper, we have presented StatCache, a novel sampled-based method to analyze data locality. Based on sparse discrete samples of memory references and measurement of their reuse distances, StatCache estimates miss ratios of fully associative caches of arbitrary sizes and generates working set graphs. This information is useful for the study of application data locality.

We have evaluated the method using the SPEC CPU2000 benchmarks, and shown that StatCache gives accurate results with a sampling rate as low as 10^{-4}. Our investigations have also shown that the difference of using fully associative caches with random or LRU replacement policies to generate working set graphs is limited. Finally, we have presented a proof-of-concept implementation capable of analyzing realistic workloads with an average slowdown of only 5.8 times, and discussed alternative implementations.

8 Future Work

There are several interesting continuations of this work. An implementation of StatCache without the limitations of code instrumentation would improve the usability. The very low sampling rate of StatCache suggests that such an implementation could have a very small execution time overhead. A method to quantify conflicts is also desirable as a complement to capacity misses. A method to relate the information to source code and data structures should probably turn out useful.

References

[1] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *High Performance Computer Architecture (HPCA-9)*, 2003.

[2] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous pro-

filing: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 1997.

[3] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of SuperComputing*, 2000.

[4] B. Buck and J. Hollingsworth. Using hardware performance monitors to isolate memory bottlenecks. In *Proceedings of Supercomputing*, 2000.

[5] M. R. Chernick. *Bootstrap Methods, A Practiotioner's Guide*. Wiley, 1999.

[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] L. DeRose, K. Ekanadham, and J. K. Hollingsworth. Sigma: A simulator infrastructure to guide memory analysis. In *Proceedings of SuperComputing*, 2002.

[8] A. Eustace and A. Srivastava. ATOM: A flexible interface for building high performance program analysis tools. In *USENIX Winter*, pages 303–314, 1995.

[9] 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.

[10] O. Grenholm. Simple and Efficient Instrumentation for the DSZOOM System. Master's thesis, School of Engineering, Uppsala University, Sweden, December 2002.

[11] R. Fowler J. Mellor-Crummey and D. Whalley. Tools for application-oriented performance tuning. In *Proceedings of the 2001 ACM International Conference on Supercomputing*, 2001.

[12] 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.

[13] AJ KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja. Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research. In *Workshop on Workload Characterization, International Conference on Computer Design (ICCD)*, 2000.

[14] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1995.

[15] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, 1994.

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

[17] J. Maebe, M. Ronsse, and K. De Bosschere. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In Eds. Charney M. and D. Kaeli, editors, *Compendium of Workshops and Tutorials. Held in conjunction with PACT'02: International Conference on Parallel Architectures and Compilation Techniques.*, September 2002.

[18] 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.

[19] M. Martonosi, A. Gupta, and T. Anderson. Memspy: Analyzing memory system bottlenecks in programs. In *ACM SIGMETRICS International Conference on Modeling of Computer Systems*, pages 1–12, 1992.

[20] M. Martonosi, A. Gupta, and T. E. Anderson. Tuning memory performance of sequential and parallel programs. *IEEE Computer*, 28(4):32–40, 1995.

[21] 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.

[22] L. Noordergraaf and R. Zak. Smp system interconnect instrumentation for performance analysis. In *Proceedings of Supercomputing*, 2002.

[23] R. Uhlig, D. Nagle, T. N. Mudge, and S. Sechrest. Trap-driven simulation with tapeworm II. In *Architectural Support for Programming Languages and Operating Systems*, pages 132–144, 1994.

[24] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995.

[25] D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. *1991 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems ACM SIGMETRICS Performance Evaluation Review*, 19(1), May 21-24, 1991.

Recent technical reports from the Department of Information Technology

2003-039	Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena: *Insights to Angluin's Learning*
2003-040	Wendy Kress: *Error Estimates for Deferred Correction Methods in Time*
2003-041	Wendy Kress: *A Compact Fourth Order Time Discretization Method for the Wave Equation*
2003-042	Gerardo Schneider: *Invariance Kernels of Polygonal Differential Inclusions*
2003-043	Kajsa Ljungberg, Sverker Holmgren, and Örjan Carlborg: *Simultaneous Search for Multiple QTL Using the Global Optimization Algorithm DIRECT*
2003-044	Dan Wallin, Henrik Johansson, and Sverker Holmgren: *Cache Memory Behavior of Advanced PDE Solvers*
2003-045	Sven-Olof Nyström: *A Polyvariant Type Analysis for Erlang*
2003-046	Martin Karlsson: *A Power-Efficient Alternative to Highly Associative Caches*
2003-047	Jimmy Flink: *Simuleringsmotor för tågtrafik med stöd för experimentell konfiguration*
2003-048	Timour Katchaounov and Tore Risch: *Interface Capabilities for Query Processing in Peer Mediator Systems*
2003-049	Martin Nilsson: *A Parallel Shared Memory Implementation of the Fast Multipole Method for Electromagnetics*
2003-050	Alexandre David: *Hierarchical Modeling and Analysis of Timed Systems*
2003-051	Pavel Krcal and Wang Yi: *Decidable and Undecidable Problems in Schedulability Analysis Using Timed Automata*
2003-052	Magnus Svärd and Jan Nordström: *Well Posed Boundary Conditions for the Navier-Stokes Equations*
2003-053	Erik Bängtsson and Maya Neytcheva: *Approaches to Reduce the Computational Cost when Solving Linear Systems of Equations Arising in Boundary Element Method Discretizations*
2003-054	Martin Nilsson: *Stability of the Fast Multipole Method for Helmholtz Equation in Three Dimensions*
2003-055	Martin Nilsson: *Rapid Solution of Parameter-dependent Linear Systems for Electromagnetic Problems in the Frequency Domain*
2003-057	Erik Berg and Erik Hagersten: *Low-Overhead Spatial and Temporal Data Locality Analysis*
2003-058	Erik Berg and Erik Hagersten: *StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis*

UPPSALA
UNIVERSITET

November 2003
ISSN 1404-3203
http://www.it.uu.se/