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. Therefore, there is a need for methods to analyze data locality and help application optimization. In this paper we present StatCache, a novel sampling-based method for performing data-locality analysis on realistic workloads. StatCache is based on a probabilistic model of the cache, rather than a functional cache simulator. It uses statistics from a single run to accurately estimate miss ratios of fully-associative caches of arbitrary sizes and generate 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 causes poor performance in many memory-intensive applications. Therefore, we need efficient methods to locate and explain cache-related performance bottlenecks.

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 data structures, while blocking is efficient in reducing capacity misses. This paper focuses on the often dominating capacity misses, and to reduce these misses we need to improve application data locality. 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 is often unbearable. Two common methods to reduce simulation time is time sampling and set sampling, but both have drawbacks [11]. Time sampling often requires very long warm-up periods, and set sampling is often very sensitive to unrepresentative set selection.

We present StatCache, a novel probabilistic approach to estimate miss ratios of fully-associative caches. 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 is that it is based on a statistic that is easy to sample. In the paper, we show that StatCache gives accurate results with an overall sampling rate as low as 10^{-4}. We present a proof-of-concept implementation with an average execution-time slowdown of only 5.8 times, 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 extended version of this paper is available as a technical report [2].

2 StatCache

StatCache is a method for analyzing run-time data locality and cache behavior. It monitors the addresses of the memory read and write operations of an application and models its data cache behavior. In this sense StatCache is comparable to trace-driven cache simulation, but StatCache is based on a probabilistic model of the cache, rather than a functional simulator. The probabilistic model uses the fact that a piece of data is less likely to reside in the cache the longer time it has been

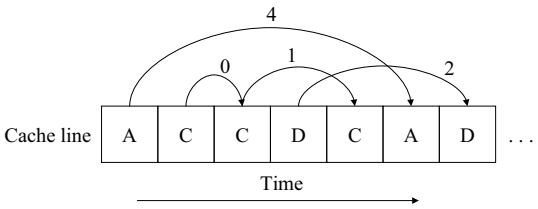

Figure 1. The figure illustrates the reuse distances. The arrows indicate reuse of cache lines, and the numbers next to the arrows are the corresponding reuse distances assigned to the memory references pointed at by the arrows.

unused.

StatCache is divided into a run-time part and a post-processing part. During run time, StatCache collects a cache-size-independent easy-to-sample data-access statistic. However, it is not possible to directly read out quantitative cache performance from the run-time statistic. Therefore, StatCache incorporates a post-processing system that applies the probabilistic cache model to the collected statistics and estimates average cache miss ratios of arbitrary-sized fully-associative caches.

2.1 Run-time Statistics

The application StatCache examines reads and writes memory. This gives a sequence of memory addresses, called address trace. The addresses in the trace are enumerated from 1 to N, where N is the total number of read and write operations the application performs. We will refer to addresses in the address trace as memory references, or simply references. StatCache considers the addresses of different data words within the same cache-line-sized piece of memory to be equal. The method currently do not consider timing effects and treats read and write operations equally.

Reuse distance is the run-time statistic that StatCache uses. It is defined as follows: Assume that the references i and j ($i < j$) access the same cache-line-sized piece of memory, A, and that there are no intermediate references to A. Then the reuse distance of reference j is $j - i - 1$, that is, the reuse distance is the number of intermediate memory references. Figure 1 illustrates this. Assume the application accesses the cache lines $A, C, C \ldots$ The arrows indicate reuse, and the numbers next to the arrows indicate the reuse distance. Note that this differs from the stack distance in that it counts all intermediate references, not just different ones. For example the stack distance of reference 6 is only 2 [20] .

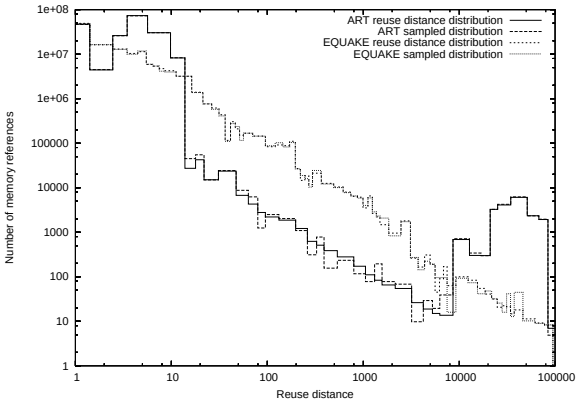

Figure 2. Reuse-distance histograms for the benchmarks art and equake. The diagram shows two graphs for each benchmark, one based on every reuse distance, and one based on sampling.

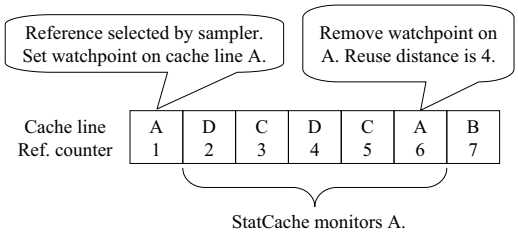

Figure 3. Sampling reuse distances.

2.2 Reuse Distance Histogram

The distribution of reuse distances can be viewed as a histogram. We call such a histogram h, and let $h(0)$ be the number of memory references with reuse distance zero, $h(1)$ the number of memory references with reuse distance one, and so on. Figure 2 shows two examples of such histograms for the SPEC benchmarks *art* and *equake*.

2.3 Sampling

The simple definition of the reuse distance makes sampling easy. The post-processing part of StatCache does not need the reuse distance of every memory reference, but base its miss-ratio estimates on reuse-distance distributions, like those shown in Figure 2. Such distributions can be estimated by measuring the reuse distance of only a small fraction of all memory references. In Figure 2, there are two versions of the reuse-distance histogram for each benchmark, one based on the reuse distance of every reference, and one approx-

imation based on sampling. The shape of the sampled and original distributions are very similar despite an overall sampling rate of only one in ten thousand.

StatCache implements sampling by monitoring randomly selected references. The run-time system in StatCache needs the following support mechanisms to do this:

A sample selector The sample selector selects random load and store instructions and start monitoring the cache-line-sized piece of data that is read or written by that instruction.

A watchpoint mechanism The watchpoint mechanism monitors the selected cache-line-sized pieces of memory and informs the StatCache run-time system when the application accesses monitored memory.

A memory reference counter The StatCache run-time system uses the memory reference counter to calculate reuse distances.

Figure 3 illustrates how these mechanisms are used. The sample selector first selects a memory reference to study and the StatCache run-time system sets a watchpoint on the corresponding cache-line-sized piece of memory. This happens at reference 1 in the example. The watchpoint then detects when the application attempts to access the same cache line again and reports to the run-time system, reference 6 in the figure. The run-time system uses the reference counter to calculate the reuse distance and records the result. It may be necessary to have several active watchpoints at a time.

2.4 Post-processing

The post-processing system in StatCache uses a probabilistic model to estimate cache miss ratios from the run-time statistic, that is, the reuse-distance histogram, described earlier.

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 $(1 - \frac{1}{L})^n$ after n cache misses. Let $f(n)$ be the probability that a cache line *does not* remain in the cache after n cache misses, i.e.

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

Next, assign a stochastic variable X_i to every memory reference. Let

$$X_i = \begin{cases} 1 \text{ if reference } i \text{ results in a cache miss} \\ 0 \text{ otherwise} \end{cases}$$

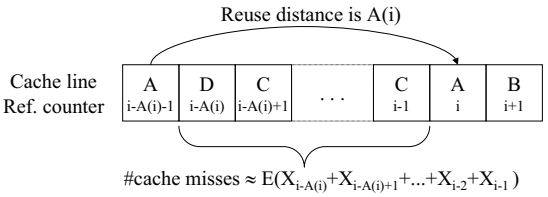

Figure 4. Estimating cache misses using reuse distance.

and let p_i be a function such that $p_i = E(X_i)$.

We want an estimate of p_i, that is, the probability that reference i causes a cache miss. First, let $A(i)$ be the reuse distance of reference i. Then use the reuse distance to find the previous reference that accessed the same cache line, this is reference $i - A(i) - 1$. Figure 4 illustrates this. The sum

$$E(X_{i-A(i)} + X_{i-A(i)+1} + \cdots + X_{i-1}) = p_{i-A(i)} + p_{i-A(i)+1} + \cdots + p_{i-1}$$

estimates the number of actual misses in the interval. Function 1 gives an estimate of p_i:

$$p_i \approx f(p_{i-A(i)} + p_{i-A(i)+1} + \cdots + p_{i-1}) \tag{2}$$

We can now sum both sides of this expression from 1 to N, which gives:

$$\sum_{i=1}^{N} p_i \approx \sum_{i=1}^{N} f(p_{i-A(i)} + p_{i-A(i)+1} + \cdots + p_{i-1}) \tag{3}$$

This expression gives a relation between all p_i, but it contains too many unknowns to deduce a formula for the overall miss ratio. Therefore, assume for now, that the overall miss ratio, R, does not change over time, which means that

$$p_i + p_{i+1} + \cdots + p_{i+j-1} = j \cdot R \tag{4}$$

where $1 \leq i \ll (i + j) \leq N$.

Note that the argument of f in expression 3 contains $A(i)$ terms. We apply assumption 4 and get $p_{i-A(i)} + p_{i-A(i)+1} + \cdots + p_{i-1} = A(i) \cdot R$. Furthermore, the left side of expression 3 becomes equal to $N \cdot R$. Thus

$$N \cdot R \approx \sum_{i=1}^{N} f\left(A(i) \cdot R\right). \tag{5}$$

Consider the sum in this expression. The number of terms with $A(i)$ equal to some constant K, is equal to the number of memory references with reuse distance

K. The run-time statistic, the histogram h, tells us that there are $h(K)$ memory references with reuse distance K. This allows expression 5 to be rewritten as

$$RN \approx h(1)f(R) + h(2)f(2R) + h(3)f(3R) + \dots \quad (6)$$

StatCache solves this equation for R and the solution is an approximation to the miss ratio. The formula is implicit, but is easy to solve with numerical methods.

Our early experiments showed that equation 6 gives accurate results for applications that satisfy the assumption that the miss ratio does not change over time, but the accuracy is poor for other applications. The solution is to approximate the varying miss ratio with a piecewise constant function. StatCache splits the execution into small time slots and generates a histogram for each slot. It then estimates the miss ratio of each time slot with equation 6 and finally calculates the average over all time slots to get an overall miss ratio.

We use a time slot length of about 200k references in this paper. This gives a rather rough histogram approximation, but our experiments showed that it was best to use short time slots. Also, the shorter the time slot is, the more miss ratios are used in the calculation of the average overall miss ratio.

2.5 Generating Working-Set Graphs

The number of cache lines, L, can be set arbitrarily when solving equation 6. Thus, a single run generates all information needed to calculate miss ratios of fully associative caches of arbitrary sizes. By solving equation 6 for different values of L, working set graphs can easily be generated. Note that the number of cache lines, L, need not be a power of 2.

2.6 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. However, the number of cold misses is usually small, so the effect can often be ignored. The good thing is that capacity misses are separated from both conflict misses and cold misses.

3 Experimental Evaluation

We have evaluated StatCache by comparing it to a functional cache simulator. The evaluation is based on trace-driven simulation, using traces generated by the Simics [17] full system simulator. It simulates a Sun UltraSPARC II workstation-like computer. We simulate both StatCache and the fully-associative caches we compare with so that the StatCache method as such may be evaluated, and exclude possible deficiencies in an implementation. To cut simulation time, we used the twenty benchmarks from the SPEC CPU2000 suite that are available with large reduced input data sets [12]. 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 matches a watchpoint, the reuse distance is added to a reuse distance histogram, like that in Figure 2, and the watchpoint is removed. The execution is split into time slots as described in section 2.4 to get accurate miss ratio estimates.

We have also simulated fully-associative caches with random replacement from 2 K Byte to 4M Byte using the traces from Simics. The miss ratios StatCache computes exclude cold misses, and to examine the effect of this, the results come in two flavors, with cold misses (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.

3.1 StatCache Accuracy

Figures 5 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).

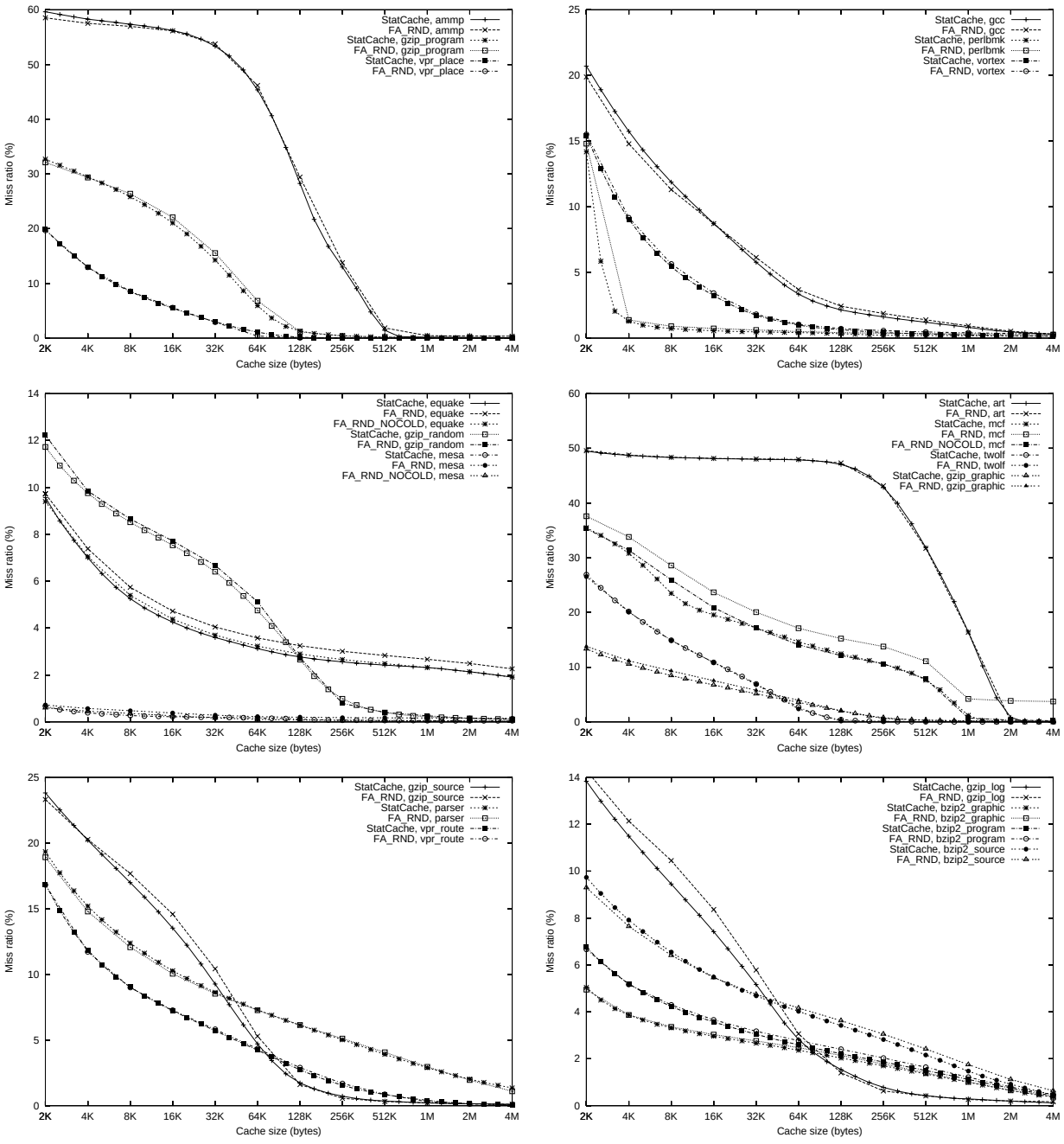

Figure 5. StatCache compared to fully-associative caches with random replacement (labeled FA_RND). The graphs show the miss ratio as a function of cache size for 20 benchmarks. StatCache accurately estimates the miss ratio of the fully-associative cache and the general behavior is well captured. The 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 include cold misses (FA_RND).

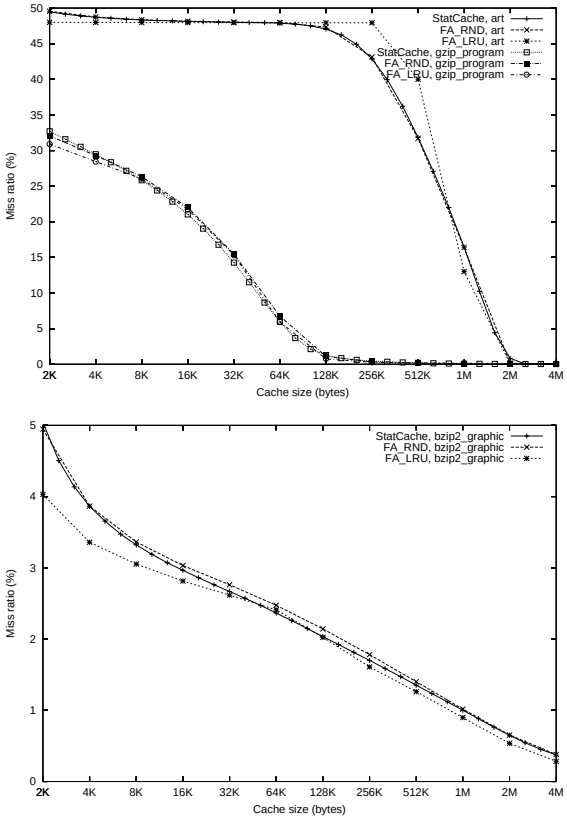

Figure 6. Examples of the difference between LRU-(FA_LRU) and random-(FA_RND) replacement policies of fully-associative caches on the benchmarks art, gzip_program, and bzip2_graphic. The graphs show that the two replacement policies usually give similar miss ratios.

3.2 Impact of Replacement Policy

Using 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 for small caches, but the difference vanishes as the cache size grows.

For *art*, LRU-replacement policy tend to give larger miss ratios for caches between 256K byte and 1M byte. This is typical of applications that traverse large amounts of data without reusing them for a long time. Compare with Figure 2, where the reuse-distance histogram of *art* has a bump for large reuse distances. For some applications, like *gzip_program*, the difference is very small.

4 Implementation

An implementation of StatCache must provide the sampling and watchpoint mechanisms, and the memory reference counter described in section 2.3.

The proof-of-concept implementation is based on code instrumentation. We use the SAIT[9] 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 is a hit, calls a watchpoint handler. The watchpoint handler removes the watchpoint, calculates and records the reuse distance, and updates the hash table.

The performance is mostly determined by the watchpoint mechanism that must check every memory access, but efficient watchpoint mechanisms may be available in hardware. Most modern processors have at least a few programmable hardware watchpoints, and the MMU and memory protection system may be used to implement watchpoints.

We used the SPEC reference inputs to evaluate the performance of the proof-of-concept implementation on ten benchmarks [1]. 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 [2].

Figure 7 shows the working-set graph of *ammp* generated by 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 5. This emphasizes the importance of performing measurements on realistic data sets.

5 Related Work

There are several ways to investigate an applications memory and cache behavior, each modelling different

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

[2] 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.

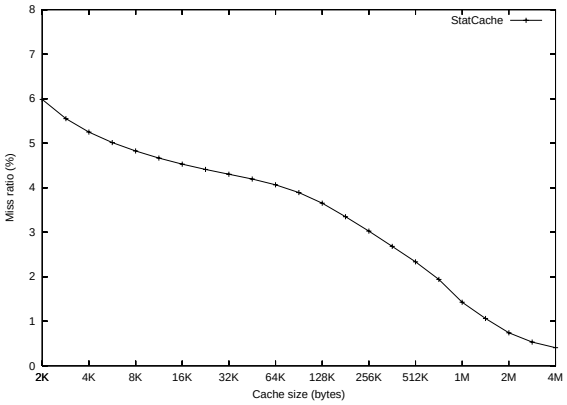

Figure 7. The output from the proof-of-concept implementation running ammp with reference input. Observe that the shape of this graph is different from the corresponding graph for ammp in Figure 5. This emphasizes the importance of performing measurements on realistic data sets.

levels of detail: full system computer simulation, instruction set simulation, user level simulation or code instrumentation, source code instrumentation, trap driven cache simulation and measurements on hardware using hardware profiling support.

Full system simulators include Simics [17] and SimOS [15]. They allow very detailed cache simulations, but suffer from large slowdown. Many tools have been built using binary code instrumentation tools like DIOTA [16], ATOM [7] or EEL [13]. Examples are SIGMA [6], CPROF [14] and MemSpy [18][19]. These tools 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 [10].

Trace sampling 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 of sampling are time sampling [22][11] [5][8] and set sampling [11] [5].

Hardware monitoring tools collect statistics from hardware and present the information in aggregated form to the user. Examples are DCPI [1], 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 [21] 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.

6 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 also indicate that the replacement policy has limited impact on the shape of the working set graph of the benchmarks in this study. 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 very fast implementations alternatives.

Acknowledgment

We would like to thank Oskar Grenholm for assistance on the instrumentation tool, Allan Gut for valuable comments on the mathematical content and Mathias Spjuth for providing the traces. This work is supported in part by Sun Microsystems, Inc., and the Parallel and Scientific Computing Institute (PSCI), Sweden.

References

[1] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 1997.

[2] E. Berg and E. Hagersten. StatCache: A probabilistic approach to efficient and accurate data locality analysis, Technical report 2003-058, Department of Information Technology, Uppsala University, November 2003.

[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] 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.

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

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

[8] 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.

[9] O. Grenholm. Simple and efficient instrumentation for the DSZOOM system. Master's thesis, School of Engineering, Uppsala University, Sweden, December 2002.

[10] 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.

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

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

[22] 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.