IT Licentiate theses
2003-15

Methods for Run Time Analysis of Data Locality

ERIK BERG

UPPSALA UNIVERSITY
Department of Information Technology

UPPSALA
UNIVERSITET

Methods for Run Time Analysis of Data Locality

BY
ERIK BERG

November 2003

DIVISION OF COMPUTER SYSTEMS
DEPARTMENT OF INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Licentiate of Philosophy in Computer Systems
at Uppsala University 2003

Methods for Run Time Analysis
of Data Locality

Erik Berg
`Erik.Berg@it.uu.se`

Division of Computer Systems
Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

`http://www.it.uu.se/`

ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden

Abstract

The growing gap between processor clock speed and DRAM access time puts new demands on software and development tools. Deep memory hierarchies and high cache miss penalties in present and emerging computer systems make execution time sensitive to data locality. Therefore, developers of performance-critical applications and optimizing compilers must be aware of data locality and maximize cache utilization to produce fast code. To aid the optimization process and help understanding data locality, we need methods to analyze programs and pinpoint poor cache utilization and possible optimization opportunities.

Current methods for run-time analysis of data locality and cache behavior include functional cache simulation, often combined with set sampling or time sampling, other regularity metrics based on strides and data streams, and hardware monitoring. However, they all share the trade-off between run-time overhead, accuracy and explanatory power.

This thesis presents methods to efficiently analyze data locality at run time based on cache modeling. It suggests source-interdependence profiling as a technique for examining the cache behavior of applications and locating source code statements and/or data structures that cause poor cache utilization. The thesis also introduces a novel statistical cache-modeling technique, StatCache. Rather than implementing a functional cache simulator, StatCache estimates the miss ratios of fully-associative caches using probability theory. A major advantage of the method is that the miss-ratio estimates can be based on very sparse sampling. Further, a single run of an application is enough to estimate the miss ratio of caches of arbitrary sizes and line sizes and to study both spatial and temporal data locality.

Acknowledgments

First of all, I would like to thank my supervisor Erik Hagersten for offering me a PhD student position, for lots of inspiring discussions, for proofreading numerous paper drafts and for guiding me towards this licentiate thesis. I would also like to thank the members of UART, Uppsala Architecture Research Team, Lars Albertsson, Martin Karlsson, Henrik Löf, Zoran Radovic, Dan Wallin and Håkan Zeffer for all interesting discussions, support on everything from LaTeX to Perl scripts, and last, but not least, creating an inspiring atmosphere to work in.

Many thanks goes to Markus Nordén for valuable discussions and providing me with with test applications. I would also like thank Oskar Grenholm for writing the instrumentation tool I have used, Mathias Spjuth for the great tracing tools, and Erik Trulsson for valuable discussions during the development of StatCache. I would also like to express gratitude to Karin Hagersten for English language support.

This work is supported in part by Sun Microsystems, Inc. and the Parallel and Scientific Computing Institute (PSCI), Sweden.

Och tack å tack å tack, å tack å tack å tack
Å tack å tack å tack å tack å tack å tack
Å tack å tack å till dig, å tack å tack å till dig
Å tack å tack å tack å tack å tack å tack

(Mel: Goddag goddag goddag, Hans Alfredsson & Tage Danielsson)

List of Appended Papers

This thesis is a summary of the following three papers. References to the papers will be made using the capital letter associated with each paper.

Paper A Erik Berg and Erik Hagersten, *SIP: Performance Tuning Through Source Code Interdependence*, Proceedings of the 8th International Euro-Par Conference (Euro-Par 2002), pages 177-186, Paderborn, Germany, August 2002. Copyright 2002, Springer Verlag. Reprinted with permission from Springer Verlag.

Paper B Erik Berg and Erik Hagersten, *StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis*, Technical report 2003-58, Department of Information Technology, Uppsala University, November 2003. Submitted for publication.

Paper C Erik Berg and Erik Hagersten, *Low Overhead Spatial and Temporal Data Locality Analysis*, Technical report 2003-57, Department of Information Technology, Uppsala University, November 2003. Submitted for publication.

1 Introduction

The performance of modern high performance processors is improving rapidly. Unfortunately, the performance of many applications does not increase with the same pace. One of the most important reasons is the relatively slow memory system. The improvement in access time and bandwidth of dynamic memories, DRAMs, have not kept pace with the processor development, and have become one of the main performance bottlenecks.

Cache memories exploit data locality to alleviate this problem. Unfortunately, many applications have data structures and/or uses algorithms that have poor locality of data and must be optimized to take full advantage of the caches. Techniques to improve data locality include tiling, loop interchange, loop merging, changing data structures and code restructuring [7, 14, 20].

Methods to analyze program data locality and cache behavior helps determine which methods to apply and which parameters to use. This analysis may be performed at compile time or run time. Compile-time methods [4, 10, 23] have the obvious advantage that the program need not be run, but many real applications are too complex to be analyzed by these methods. Run-time analysis can on the other hand handle most types of applications. Available methods and tools collect statistics either from hardware counters [1, 2, 3, 8, 12, 21] or by simulating caches, possibly in combination with code analysis[9, 11, 15, 17, 18, 19].

Programmers performing manual tuning, advanced compilers and run-time optimizers can benefit from data-locality analysis [1, 5, 6, 15, 18]. For effective manual tuning, the author believes that data-locality analysis should be integrated in profiling tools that identifies bottlenecks in source code and identifies poor data-structure layouts. The run-time methods requires that the application is executed, and can not be directly integrated in a compiler. However, advanced compilers have frameworks for feedback of for example basic block profiles, and data locality should also be implementable in such frameworks. There are also interesting methods for dynamic optimization of running codes based on application data behavior and access patterns[6].

This thesis presents methods and prototype tools for run-time data-locality analysis based on cache modeling. Paper A presents a simulator-based prototype tool, SIP, for helping software developers to find performance bottlenecks caused by poor cache behavior. The tool gives detailed information about performance bottlenecks caused by poor cache behavior and tells where in the source code the problems are and which data structures that are involved. It has been used to profile advanced numeric codes[22].

Paper B and C present two versions of a novel probabilistic cache model. Current analysis of cache behavior is usually based on functional simulation of caches.

In the probabilistic approaches described in this thesis, a cache is instead modeled statistically. The probabilistic cache models do not contain any exact cache states like cache tags or valid bits and can not tell exactly whether the cache contains valid data for any specific address, but rather give the probability of hit or miss. To reduce the run-time overhead, the models are also designed to base their analysis on information that can easily be collected at run time using very sparse sampling.

2 Data Locality Profiling

Paper A describes a prototype tool, SIP, for cache behavior profiling. It introduces source interdependence profiling, a concept for studying the cache interaction and interdependence between statements in a program. SIP maps cache miss rate information to source code and data structures, and present detailed data locality information. It collects interdependence information by tracking every cache line during their tenure in the cache and records statistics about spatial and temporal usage. Temporal usage is presented as the number of times data in a cache line is reused and spatial use as the fraction of the data in the cache line that is ever read or written before the cache line gets replaced. The tool is implemented using a full system simulator [16]. This enables it to run unmodified code, and can be used to study programs written in both C and Fortran. The tool uses a graphical user interface to present the profile information.

The paper discusses a case study of the SPEC 183.equake benchmark, an earth quake simulator, and shows that it can be optimized to run 25 percent faster as 32-bit program on an UltraSPARC II based computer, and 59 percent faster as 64-bit program. Especially the spatial use metric turned out to be useful and exposed a performance degrading memory allocation in the earth quake simulator. The major problem with the prototype implementation was its poor performance caused by the simulator, which prevented it to be extensively tested on realistic workloads. The design decision to use a simulator was however made because we wanted to test the method of collecting statistics and see if it would prove useful, rather than implementing a full-blown tool.

3 Probabilistic Modeling

To analyze the data locality of an application, we can observe the sequence of memory addresses of the read and write operations that are executed. We number these memory addresses and call them memory references. The traditional way to model cache behavior is to feed the memory references to a functional cache simulator and count cache hits and misses. Unfortunately, this results in a high runtime overhead, especially if we want information about many cache sizes in a single run. Therefore, many approaches to data-locality and cache-behavior analysis are based on different sampling techniques, such as set sampling and time sampling. Set sampling reduces the simulation work by only simulating a subset of all sets in a set-associative cache. Time sampling simulates a complete cache, and reduces the execution overhead by feeding the simulator with short continuous subsequences of the memory references. However, unrepresentative samples and cold start effects can impair accuracy and the slowdown is still considerable for simulating many cache sizes in a single run.

In paper B, we introduce a novel cache modeling and sampling method, StatCache, for estimating the miss ratios of fully-associative caches. Rather than modeling the exact functionality of a cache, StatCache takes a probabilistic view of the cache. It is inspired by the observation that the probability that a piece of data remains in the cache depends on the number of cache misses that have occurred since it was last accessed [13]. The longer a piece of data has been untouched, the higher is the probability that it has been replaced by other data.

The reuse distance is the number of memory references between two consecutive memory references to the same piece of data. In paper B, we derive a formula for estimating the probability that a specific memory reference will cause a cache miss based on the miss probabilities of previous memory references, and show that the distribution of the data-reuse distances of an application can be used to estimate the average miss ratio of the entire application. Since StatCache only needs the distribution of the reuse distances, it can base its miss-ratio estimates on sparse samples, rather than measure every reuse distance. An important advantage of the reuse distance that StatCache uses is that it is easy to measure. A simple memory-reference counter, often available in hardware, and a method to monitor references to a few cache-line-sized pieces of memory is all that is needed.

The paper evaluates the accuracy of the method on twenty SPEC benchmarks. We show in the paper that the accuracy of the method is high and that it predicts different cache behaviors well, based on reuse-distance measurement of only one reference in ten thousand. A proof-of-concept implementation, based on code instrumentation, is also described in the paper. The slowdown is only 5.8 times on average despite that it is a simple first implementation.

4 DLTune: Spatial and Temporal Analysis

Paper C describes a more advanced prototype, DLTune, based on a method similar to that described in paper B. Both spatial and temporal analysis is possible in a single run because the tool can estimate miss ratios of caches with both varying cache size and cache line size. The paper describes measures of temporal and spatial locality that can be calculated from the information produced by the tool. We show examples of how the tool can be used and apply the tool to the same earth quake simulator as in paper A and demonstrate that we can find the same spatial-locality problem as before. The tool can also separate the data-locality information for different data structures in the program. A refinement of the sampling method in Paper B allows this new tool to use a sampling rate that is only 10^{-7} on long-running applications and the average slowdown is reduced to 4.3 times.

5 Summary of Contributions

This thesis includes novel research in the following areas:

- SIP: A paradigm for source-interdependence profiling.
- A prototype implementation of SIP.
- StatCache: A novel probabilistic approach for cache-miss-ratio estimation based on very sparse discrete sampling.
- A proof-of-concept implementation of StatCache.
- An enhanced version of StatCache that uses bursty sampling to further reduce the sampling rate to one memory access in ten millions for long-running applications.
- DLTune: A prototype implementation of the enhanced sampling method.
- Case studies of data locality in SPEC benchmarks and how the information collected by the prototype tools described in the thesis can be used to optimize performance.

6 Future Work

The good accuracy of StatCache, and the interesting results from the DLTune prototype encourage further research. The plans for future work include continued work with StatCache to improve the implementation to allow more experiments on a variety of applications. Possible areas of use that are left to be explored include automatic code optimization, possibly within a compiler frame-work. The data-locality analysis provided by StatCache may also be used to direct software prefetching.

It is tempting to use hardware support for counting memory references and monitoring memory to reduce run-time overhead further and remove the need for code instrumentation.

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] 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.

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

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

[5] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 191–202, 2001.

[6] T. M. Chilimbi. Dynamic hot data stream prefetching for general-purpose programs. In *PLDI*, 2002.

[7] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[8] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, 1997.

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

[10] 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.

[11] A. Goldberg and J. Hennessy. MTOOL: A method for isolating memory bottlenecks in shared memory multiprocessor programs. In *Proceedings of the International Conference on Parallel Processing*, pages 251–257, 1991.

[12] M. Itzkowitz, B.J.N. Wylie, C. Aoki, and N. Kosche. Memory profiling using hardware counters. In *Proceedings of Supercomputing*, 2003.

[13] M. Karlsson and E. Hagersten. Timestamp-Based Selective Cache Allocation. In *Proceedings of the Workshop on Memory Performance Issues, held in conjunction with the 28th International Symposium on Computer Architecture (ISCA28)*, Göteborg, Sweden, June 2001.

[14] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 346–357, 1997.

[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] 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.

[17] 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.

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

[19] M. Martonosi, D. Ofelt, and M. Heinrich. Integrating performance monitoring and communication in parallel computers. In *Measurement and Modeling of Computer Systems*, pages 138–147, 1996.

[20] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

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

[22] Markus Nordén, Malik Silva, Sverker Holmgren, Michael Thuné, and Richard Wait. Implementation issues for high performance CFD. In *International Information Technology Conference - 2002*, 2002.

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

Paper A

SIP: Performance Tuning Through Source Code Interdependence

Reprinted from

Proceedings of the 8th International Euro-Par Conference (Euro-Par 2002),
pages 177-186, Paderborn, Germany, August 2002.
Reprinted with permission from Springer Verlag.

SIP: Performance Tuning Through Source Code Interdependence

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

November 25, 2003

Abstract

The gap between CPU peak performance and achieved application performance widens as CPU complexity, as well as the gap between CPU cycle time and DRAM access time, increases. While advanced compilers can perform many optimizations to better utilize the cache system, the application programmer is still required to do some of the optimizations needed for efficient execution. Therefore, profiling should be performed on optimized binary code and performance problems reported to the programmer in an intuitive way. Existing performance tools do not have adequate functionality to address these needs.

Here we introduce *source interdependence profiling*, SIP, as a paradigm to collect and present performance data to the programmer. SIP identifies the performance problems that remain after the compiler optimization and gives intuitive hints at the source-code level as to how they can be avoided. Instead of just collecting information about the events directly caused by each source-code statement, SIP also presents data about events from some interdependent statements of source code.

A first SIP prototype tool has been implemented. It supports both C and Fortran programs. We describe how the tool was used to improve the performance of the SPEC CPU2000 183.equake application by 59 percent.

1 Introduction

The peak performance of modern microprocessors is increasing rapidly. Modern processors are able to execute two or more operations per cycle at a high rate. Unfortunately, many other system properties, such as DRAM access times and cache sizes, have not kept pace. Cache misses are becoming more and more expensive. Fortunately, compilers are getting more advanced and are today capable of doing many of the optimizations required by the programmer some years ago, such as blocking. Meanwhile, the software technology has matured, and good programming practices have been developed. Today, a programmer will most likely aim at, first, getting the correct functionality and good maintainability; then, profile to find out where in the code the time is spent; and, finally optimizing that fraction of the code. Still, many applications spend much of their execution time waiting for slow DRAMs.

Although compilers evolve, they sometimes fail to produce efficient code. Performance tuning and debugging are needed in order to identify where an application can be further optimized as well as how it should be done. Most existing profiling tools do not provide the information the programmer needs in a straightforward way. Often the programmer must have deep insights into the cache system and spend a lot of time interpreting the output to identify and solve possible problems.

Profiling tools are needed to explain the low-level effects of an application's cache behavior in the context of the high level language. This paper describes a new paradigm that gives straightforward aid to identify and remove performance bottlenecks. A prototype tool, implementing a subset of the paradigm, has proven itself useful to understand home-brewed applications at the Department of Scientific Computing at Uppsala University. In this paper we have chosen the SPEC CPU2000 183.equake benchmark as an example.

The paper is outlined as follows. Section 2 discusses the ideas behind the tool and general design considerations. Section 3 gives the application writer's view of our first prototype SIP implementation; Section 4 demonstrates how it is used for tuning of equake. Section 5 describes the tool implementation in more detail, section 6 compares SIP to other related tools, before the final conclusion.

2 SIP Design Considerations

The semantic gap between hardware and source code is a problem of application tuning. Code-centric profilers, which for example present the cache miss rate per source-code statement, reduces this gap, but the result can be difficult to interpret. We have no hints as to why the misses occurred. High cache miss ratios are often not due to one single source code statement, but depend on the way different statements interact, and how well they take advantage of the particular data layout used. Data-centric profilers instead collect information about the cache utilization for different data structures in the program. This can be useful to identify a poorly laid out, or misused, data structure. However, it provides little guidance to exactly where the code should be changed.

We propose using a profiler paradigm that presents data based on the interdependence between source code statements: *Source Interdependence Profiler*, SIP. SIP is both code-centric, in that statistics are mapped back on the source code, and data-centric, in that the collected statistics can be subdivided for each data structure accessed by a statement. The interdependence information for individual data structures accessed by a statement tells the programmer which data structures that may be restructured or accessed in a different way to improve performance.

The interdependence between different memory accesses can be either positive or negative. *Positive cache interdependence*, i.e., a previously executed statement has touched the same cache line, can cause a cache hit; *negative cache interdependence*, i.e., a more recent executed statement has touched a different cache line indexing to the same cache set and causing it to be replaced, may cause a cache miss. A statement may be interdependent with itself because of loop constructs or because it contains more than one access to the same memory location.

To further help the programmer, the positive cache interdependence collected during a cache line's tenure in the cache is subdivided into spatial and temporal locality. The spatial locality tells how large fraction of the cache line was used before eviction, while the temporal locality tells how many times each piece of data was used on average.

3 SIP Prototype Overview

The prototype implementation works in two phases. In the first phase, the studied application is run on the Simics [10] simulator. A cache simulator and a statistics collector is connected to the simulator. During the execution of the studied application, cache events are recorded and associated with load and store instructions in the binary executable. In the second phase, an analyzer summarizes the gathered information and correlates it to the studied source code. The output from the analyzer consists of a set of HTML files viewable by a standard browser. They contain the source code and the associated cache utilization.

Figure 1 shows a sample output from the tool. The browser shows three panes. To the left is an index pane where the source file names and the data structures are presented, the upper right pane shows the source code, and the lower right contains the results of the source-interdependence analysis. A click on a source file name in the index pane will show the content of the original source file with line numbers in the source pane. It will also show estimated relative execution costs to the left of the last line of every statement in the file. Source statements with high miss rates or

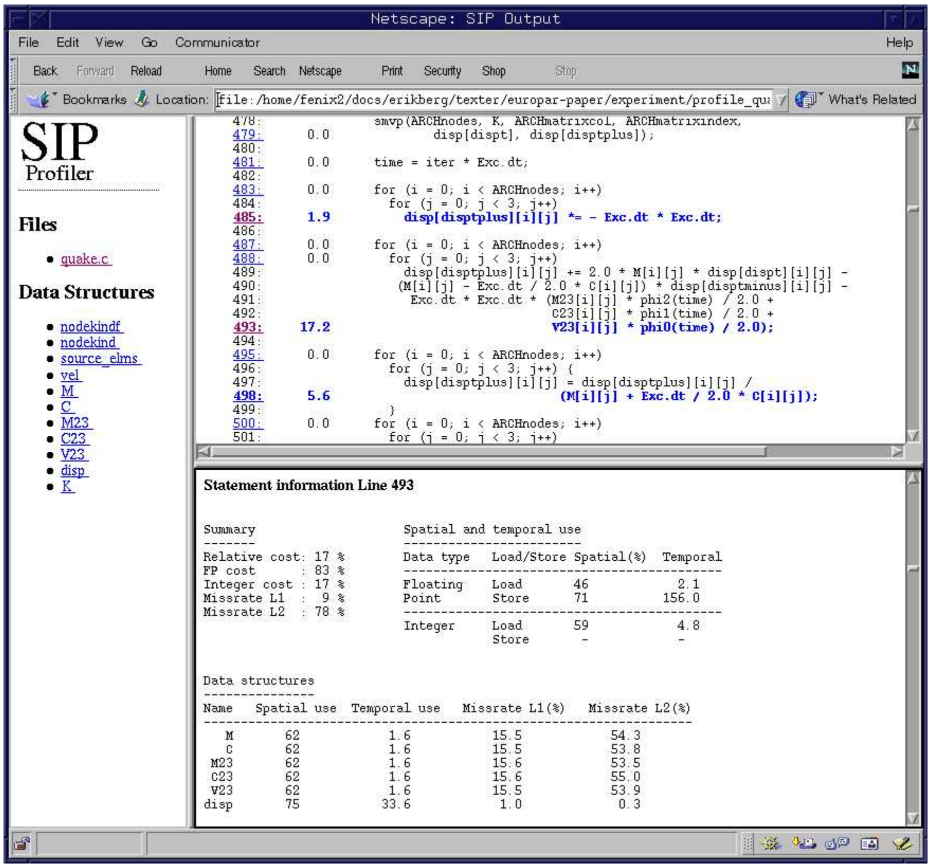


Figure 1: A screen dump from experiments with SPEC CPU2000 183.equake, 32 bit binary on UltraSPARCII. It shows the index pane (left), source pane (right) and profile information pane (bottom). It shows that the application exhibits poor spatial locality (46 percent) and temporal locality (2.1 times) for floating point loads.

execution times are colored and boldfaced. The source-interdependence analysis results for a statement can be viewed by clicking on the line number of the last line of the statement. It will show in the left lower pane in the following three tables:

- **Summary**

  A summary of the complete statement. It shows the estimated relative cost of the statement as a fraction of the total execution time of the application, the fraction of load/store cost caused by floating point and integer accesses, and miss rates for first- and second-level caches.

- **Spatial and Temporal Use**

  Spatial and temporal use is presented for integer and floating point loads and stores. The Spatial and Temporal use measures are chosen to be independent from each other to simplify the interpretation.

  - **Spatial use**
    Indicates how large fraction of the data brought into cache that is ever used. It is the percentage, on average, of the number of bytes allocated into cache by this statement that are ever used before evicted. This includes used by this same statement again, e.g. in next iteration of a loop, or used by another statement elsewhere in the program.
  - **Temporal use**
    The average number of times data is reused during its tenure in the cache. First touch is not counted, i.e. a temporal use equal to zero indicates that none of the data is not touched more than once before it is evicted. Data that is never touched is disregarded, and therefore this measure does not depend on the spatial use.

- **Data Structures:**

  Miss ratios, spatial use and temporal use are presented for the individual data structures, or arrays, accessed by the statement.

This prototype SIP implementation does not implement the explicit pointers to other statements where data is reused, but only the implicit interdependence in spatial and temporal use. We anticipate that future enhancements of the tool will include the explicit interdependencies.

4 Case Study: SPEC 183.equake

A case study shows how SIP can be used to identify and help understanding of performance problems. We have chosen the 183.equake benchmark from the SPEC [15] CPU2000 suite. It is an earthquake-simulator written in C. First, SIP was used to identify the performance bottlenecks in the original[1] application and examine their characteristics. Figure 1 shows a screen dump of the result. The statement on lines 489-493 accounts for slightly more than 17 percent of the total execution time. Click on “493”, and the browser will show the statement information in the lower

[1]In the prototype, the main function must be instrumented with a start call to tell SIP that the application has started. Recognizable data structures must also be instrumented. For heap-allocated data structures, this can be done automatically.

pane as in the figure. As can be seen under Summary, the cost of floating-point loads and stores is large. Miss rates are also large, especially in the Level 2 cache.

4.1 Identifying Spatial Locality Problems

The spatial use shows poor utilization of cached data. Floating-point loads show the worst behavior. As can be seen in the lower right pane under "Spatial and temporal use", not more than 46 percent of the floating-point data fetched into cache by loads in this statement are ever used. Floating-point store and integer loads behave better, 71 and 59 percent respectively. The information about individual data structures, in bottom table of the same pane, points in the same direction. All but one, the array disp, have only 62 percent spatial use. When examining the code, the inner-most loop, beginning on line 488, corresponds to the last index of the data accesses on lines 489 - 492. This should result in good spatial behavior and contradicts the poor spatial percentage reported by the tool.

These results caused us to take a closer look at the memory layout. We found a problem in the memory allocation function. The data structure in the original code is a tree, where the leafs are vectors containing three doubles each. The memory allocation function does not allocate these vectors adjacent to each other, but leaves small gaps between them. Therefore not all of the data brought into the cache are ever used, causing the poor cache utilization.

A simple modification of the original memory-allocation function substantially increases performance. The new function allocates all leaf vectors adjacent to each other and the SIP tool shows that the spatial use of data improves. The speedups caused by the memory-allocation optimization are 43 percent on a 64-bit (execution time reduced from 1446s to 1008s) and 10 percent on a 32-bit executable. The probable reason of the much higher speedup on the 64-bit binary is that the larger pointers cause larger gaps between the leafs in the original memory allocation. The SIP tool also revealed other code spots that benefit from this optimization. Therefore the speedup of the application is larger than the 17 percent execution cost of the statement on lines 489-493. A matrix-vector multiplication especially benefits by the above optimization. All speedup measurements were conducted with a Sun Forte version 6.1 C compiler and a Sun E450 server with 16KB level 1 data cache, 4MB unified level 2 cache and 4GB of memory, running SunOS 5.7. Both 64- and 32-bit executables were created with the *-fast* optimization flag. All speed gains were measured on real hardware.

4.2 Identifying Temporal Problems

The temporal use of data is also poor. For example, Figure 1 shows that floating-point data fetched into the cache from the statement are only reused 2.1 times on average. The code contains four other loop nests that access almost the same data structures as the loop nest on lines 487-493. They are all executed repeatedly in a sequence. Because the data have not been reused more, the working sets of the loops are too large to be contained in the cache. Code inspection reveals that loop merging is possible. Profiling an optimized version of the program with the loops merged shows that the data reuse is much improved. The total speedups with both this and the previous memory allocation optimizations are 59 percent on a 64-bit and 25 percent on a 32-bit executable.

5 Implementation Details

The prototype implementation of SIP is based on the Simics full-system simulator. Simics[10] simulates the hardware in enough detail to run an unmodified operating system and, on top of that, the application to be studied. This enables SIP to collect data non-intrusively and to take operating-system effects, such as memory-allocation and virtual memory system policies, into account. SIP is built as a module of the simulator, so large trace files are not needed. The tool can profile both Fortran and C code compiled with Sun Forte compilers and can handle highly optimized code. As described earlier, the tool works in two phases, the collecting phase and the analyzing phase.

5.1 SIP Collecting Phase

During the collecting phase, the studied application is run on Simics to collect cache behavior data. A memory-hierarchy simulator is connected to Simics. It simulates a multilevel data-cache hierarchy. The memory-hierarchy simulator can be configured for different cache parameters to reflect the characteristics of the computer, for which the studied application is to be optimized. The parameters are cache sizes, cache line sizes, access times, etc. The slowdown of the prototype tool's analyzing phase is around 450 times, mostly caused by the simulator, Simics.

The memory hierarchy reports every cache miss and evicted data to a statistics collector. Whenever some data is brought to a higher level of the cache hierarchy, the collector starts to record the studied application's use of it. When data are evicted from a cache, the recorded information is associated with the instruction that originally caused the data to be allocated into the cache. All except execution count and symbol reference are kept per cache level. The information stored for each load or store machine instruction includes the following:

- **Execution count**

  The total number of times the instruction is executed.

- **Cache misses**

  The total number of cache misses caused by the instruction.

- **Reuse count**

  The reuse count of one cache-line-sized piece of data is the number of times it is touched from the time it is allocated in the cache until it is evicted. Reuse count is the sum of the reuse counts of all cache-line-sized pieces of data allocated in the cache.

- **Total spatial use**

  The sum of the spatial use of all cache-line-sized pieces of data allocated in cache. The spatial use of one cache line-sized-piece of data is the number of different bytes that have been touched from the time it is allocated in cache until it is evicted.

- **Symbol reference**

  Each time a load or store instruction accesses memory, the address is compared to the address ranges of known data structures. The addresses of the data structures comes from instrumenting the source code. If a memory-access address matches any known data structure, a reference to that data structure is associated with the instruction PC. This enables the tool to relate caching information with specific data structures.

5.2 SIP Analyzing Phase

The analyzer uses the information from the statistics collector and produces the output. First, a mapping from machine instructions to source statements is built. This is done for every source file of the application. Second, for each source code statement, every machine instruction that is related to it is identified. Then, the detailed cache behavior information can be calculated for every source statement; and finally, the result is output as HTML files.

SIP uses compiler information to relate the profiling data to the original source code. To map each machine instruction to a source-code statement, the analyzer reads the debugging information [16] from the executable file and builds a translation table between machine-instruction addresses and source-code line numbers. The machine instructions are then grouped together per source statement. This is necessary since the compiler reorganizes many instructions from different source statements during optimization and the tool must know which load and store instructions that belongs to any source statement. The accurate machine-to-source-code mapping generated by Sun Forte C and F90 compilers makes this grouping possible. It can often be a problem to map optimized machine code to source code, but in this case it turned out to work quite well.

Derived measures are calculated at source-statement level. The information collected for individual machine instructions are summarized over their respective source-code statements, i.e. total spatial use for one statement is the sum of the total spatial uses of every load and store instruction that belongs to that statement. Reuse count is summarized analogous. To calculate the information that is presented in the table "Spatial and temporal use" in Figure 1, instructions are further subdivided into integer load, integer store, floating point load and floating point store for each source statement. For example, the total spatial use for floating-point load of one statement is the sum of the total spatial uses of every floating-point load instruction that belongs to that statement. The spatial use for a statement is calculated as:

$$Spatial\ use(\%) = 100 \cdot \frac{total\ spatial\ use\ of\ the\ statement}{\#cache\ misses\ of\ the\ statement \cdot cache\ line\ size}$$

Temporal use is calculated as:

$$Temporal\ use = \frac{reuse\ count\ of\ the\ statement}{total\ spatial\ use\ of\ the\ statement} - 1$$

The output is generated automatically in HTML format. It is easy to use and it does not need any specialized viewer. SIP creates two output files for each source file, one that contains the source code with line numbers, and one that contains the detailed cache information. It also produces a main file that sets up frames and links to the other files.

6 Related Work

Source-code interdependence can be investigated at different levels. Tools that simply map cache event counts to the source code do not give enough insights in how different parts of the code interact. Though useful, they fail to fully explain some performance problems. Cacheprof [14] is a tool that annotates source-code statements with the number of cache misses and the hit-and-miss ratios. It is based on assembly code instrumentation of all memory access instructions. For every memory access, a call to a cache simulator is inserted.

MemSpy [11] is based on the tango [6] simulator. For every reference to dynamically allocated data, the address is fed to a cache simulator. It can be used for both sequential and parallel applications. The result is presented at the procedure and data-structure level and indicates whether the misses were caused by communication or not. The FlashPoint tool [12] gathers similar information using the programmable cache-coherence controllers in the FLASH multiprocessor computer. CPROF [8] uses a binary executable editor to insert calls to a cache simulator for every load and store instruction. It annotates source code with cache-miss ratios divided into the categories of compulsory, conflict and capacity. It also gives similar information for data-structures. It does not investigate how different source statements relate to each other through data use, except for the implicit information given by the division into conflict and capacity. The full system simulator SimOS[9] has also been used to collect similar data and to optimize code. MTOOL[5] is a tool that compares estimated cycles due to pipeline stalls with measurements of actual performance. The difference is assumed to be due to cache miss stalls. Buck and Hollingsworth [2] present two methods for finding memory bottlenecks; counter overflow and n-way search based on the number of cache misses to different memory regions.

DCPI [1] is a method to get systemwide profiles. It collects information about such things as cache misses and pipeline stalls and maps this information to machine or source code. It uses the ProfileMe[3] hardware mechanism in the Alpha processor to accurately annotate machine instructions with different event counters, such as cache misses and pipeline stalls. The elaborate hardware support and sampling of nearby machine instructions can find dependencies between different machine instructions, but the emphasis is on detailed pipeline dependencies rather than memory-system interaction. SvPablo [4] is a graphical viewer for profiling information. Data can be collected from different hardware counters and mapped to source code. The information is collected by instrumenting source-code with calls to functions that read hardware counters and records there values. Summaries are produced for procedures and loop constructs.

MHSIM[7] is the tool that is most similar to SIP. It is based on source-code instrumentation of Fortran programs. A call to a memory-hierarchy simulator is inserted for every data access in the code. It gives spatial and temporal information at loop, statement and array-reference levels. It also gives conflict information between different arrays. The major difference is that it operates at source-code level and therefore gives no information as to whether the compiler managed to remove any performance problems. The temporal measure in MHSIM is also less elaborate. For each array reference, it counts the fraction of accesses that hit previously used data.

7 Conclusions and Future Work

We have found that source-code interdependence profiling is useful to optimize software. In a case study we have shown how the information collected by SIP, Source code Interdependence Profiling, can be used to substantially improve an application's performance. The mechanism to detect code interdependencies increases the understanding of an application's cache behavior. The comprehensive measures of spatial and temporal use presented in the paper also proved useful. It shows that further investigation should prove profitable.

Future work includes adding support to relate different pieces of code to each other through their use of data. Further, we intend to reduce the tool overhead by collecting the information by assembly code instrumentation and analysis. We also plan to incorporate this tool into DSZOOM [13], a software distributed shared memory system.

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

[3] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, 1997.

[4] L. DeRose and D. Reed. Svpablo: A multi-language architecture-independent performance analysis system. In *10th International Conference on Performance Tools*, pages 352–355, 1999.

[5] A. Goldberg and J. Hennessy. MTOOL: A method for isolating memory bottlenecks in shared memory multiprocessor programs. In *Proceedings of the International Conference on Parallel Processing*, pages 251–257, 1991.

[6] S. Goldschmidt H. Davis and J. Hennessy. Tango: A multiprocessor simulation and tracing system. In *Proceedings of the International Conference on Parallel Processing*, 1991.

[7] 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.

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

[9] 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.

[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] 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.

[12] M. Martonosi, D. Ofelt, and M. Heinrich. Integrating performance monitoring and communication in parallel computers. In *Measurement and Modeling of Computer Systems*, pages 138–147, 1996.

[13] Z. Radovic and E. Hagersten. Removing the overhead from software-based shared memory. In *Proceedings of Supercomputing 2001*, November 2001.

[14] J. Seward. The cacheprof home page
`http://www.cacheprof.org/`.

[15] SPEC. Standard performance evaluation corporation
`http://www.spec.org/`.

[16] Sun. *Stabs Interface Manual, ver.4.0*. Sun Microsystems, Inc, Palo Alto, California, U.S.A., 1999.

Paper B

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

Reprinted from

Technical report 2003-58
Department of Information Technology
Uppsala University
Uppsala, Sweden, November 2003.

Submitted for publication.

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.

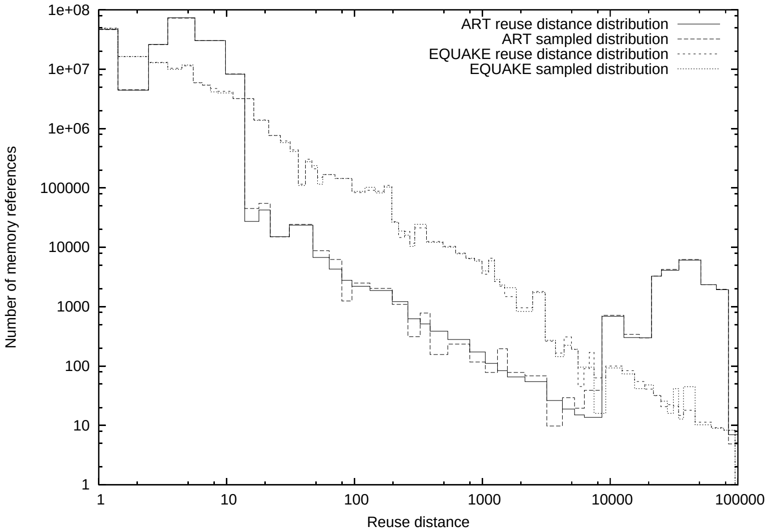


Figure 2: The diagram shows the reuse-distance distribution, i.e. the number of memory references as a function of reuse distance, for the SPEC benchmarks art and equake. It shows the distributions of complete memory access traces and of sampled traces. The sampled distributions are very similar to their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

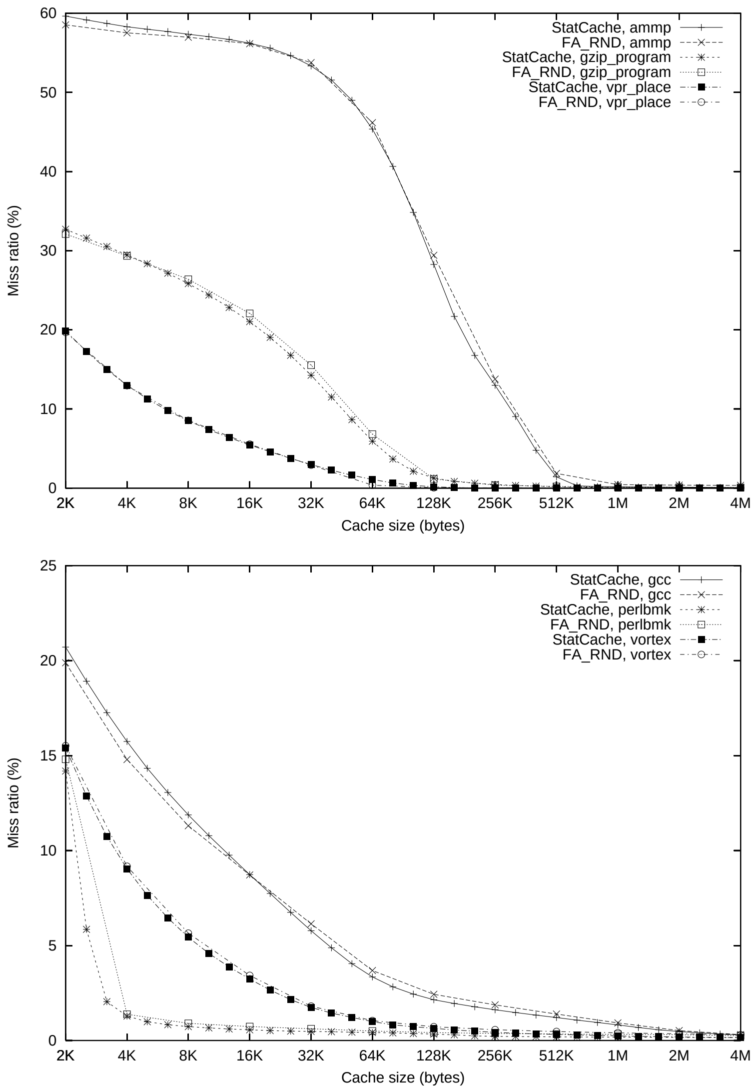


Figure 3: StatCache compared to fully-associative caches with random replacement (labeled FA_RND). The graphs show the miss ratio as a function of cache size for 20 benchmarks. StatCache accurately estimates the miss ratio of the fully-associative cache and the general behavior is well captured. The 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.

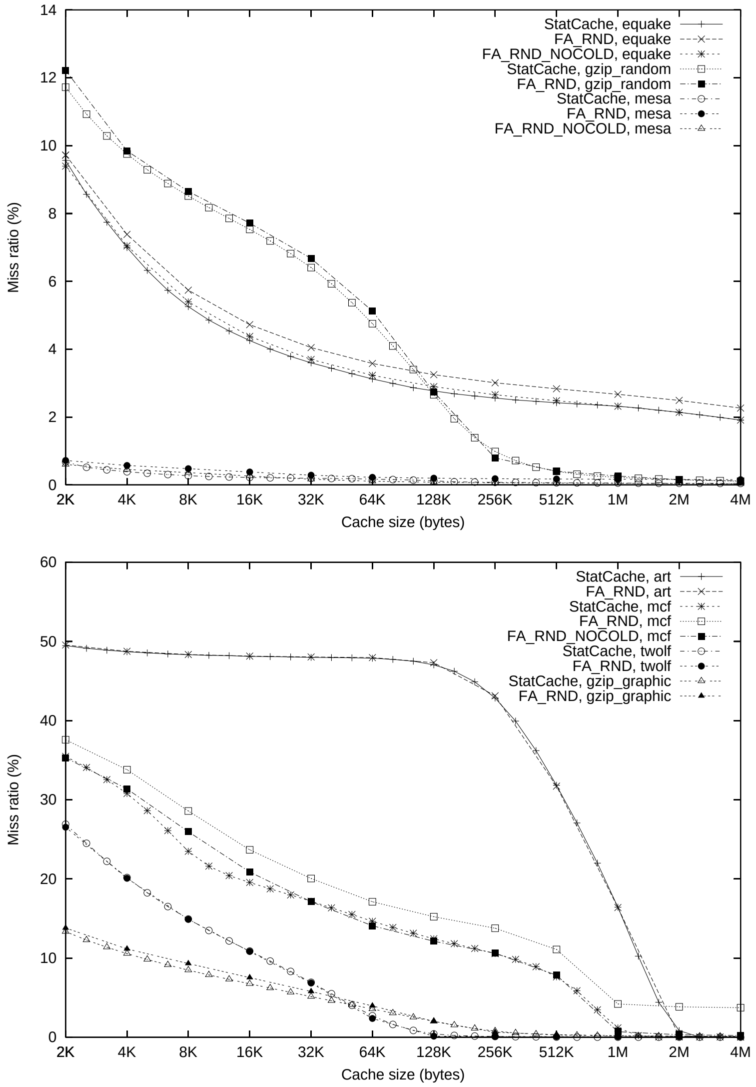


Figure 4: StatCache compared to fully-associative caches with random replacement (labeled FA_RND). The figure is a continuation from previous page.

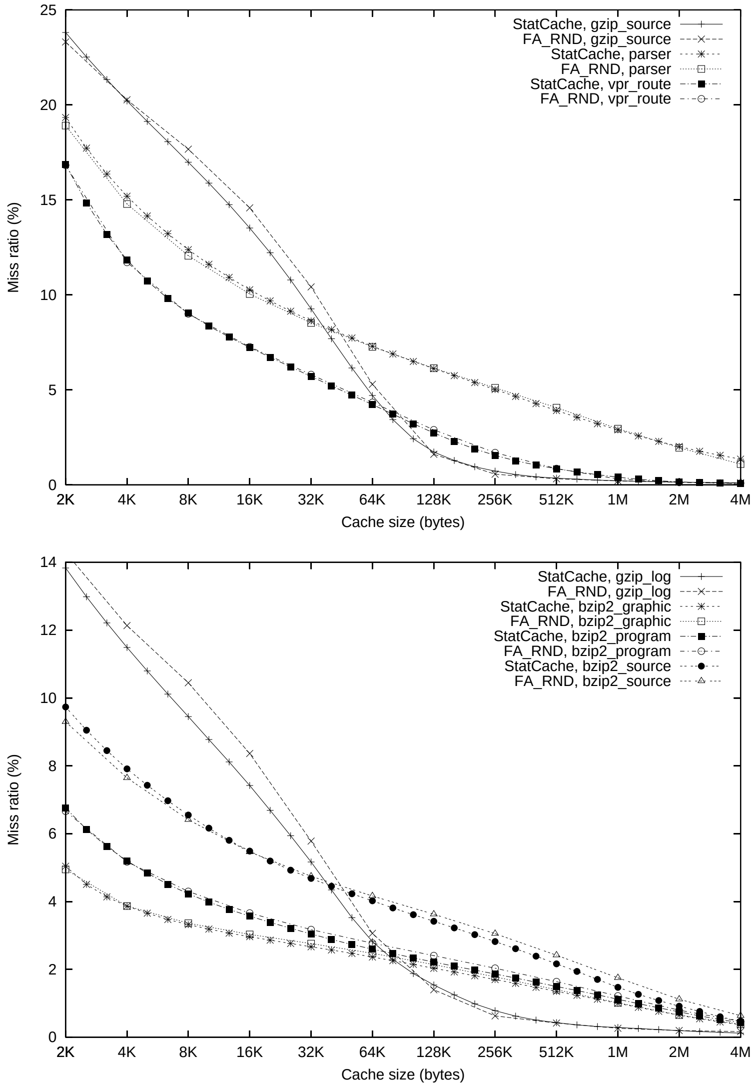


Figure 5: StatCache compared to fully-associative caches with random replacement (labeled FA_RND). The figure is a continuation from previous page.

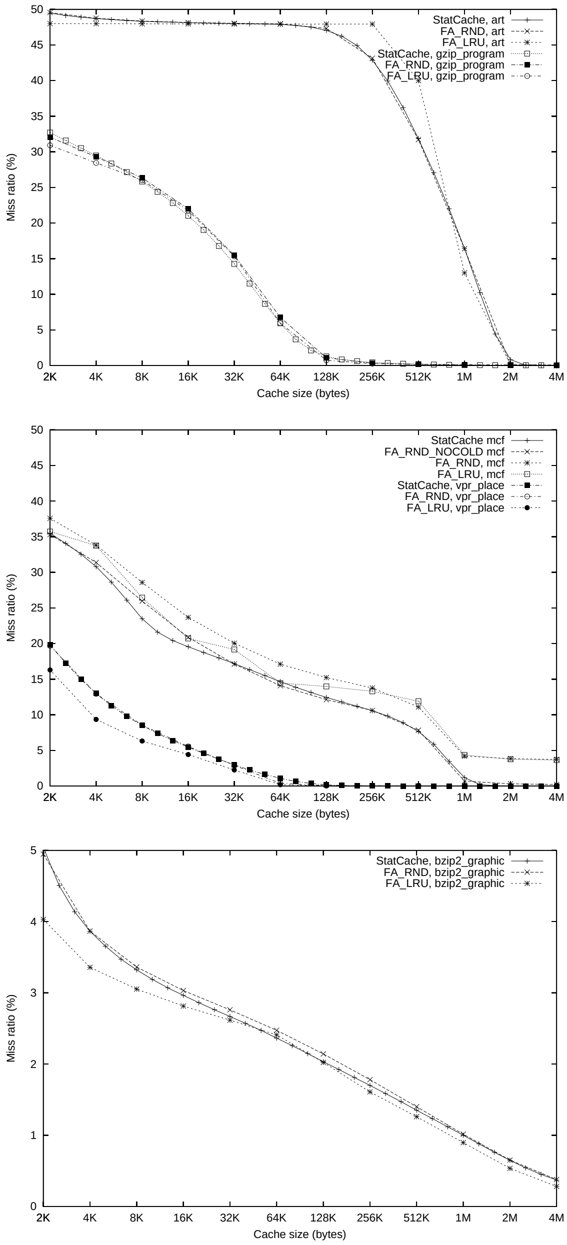


Figure 6: Examples of the difference between LRU-(FA_LRU) and random-(FA_RND) replacement policies of fully-associative caches on the benchmarks art, gzip_program, mcf, vpr_place, and bzip2_graphic. The graphs show that the two replacement policies usually give similar miss ratios.

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

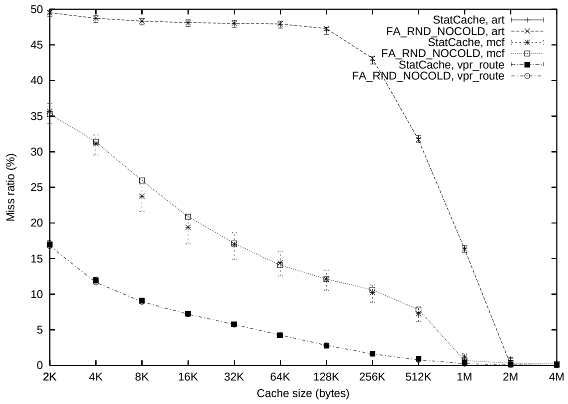
Set sampling using highly-associative caches is an alternative for studying data locality, but the accuracy is a problem. Figure 8 shows set sampling compared to simulated set-associative caches. The solid line shows the miss ratios of simulated 8-way caches, and the points show set sampling estimates 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.

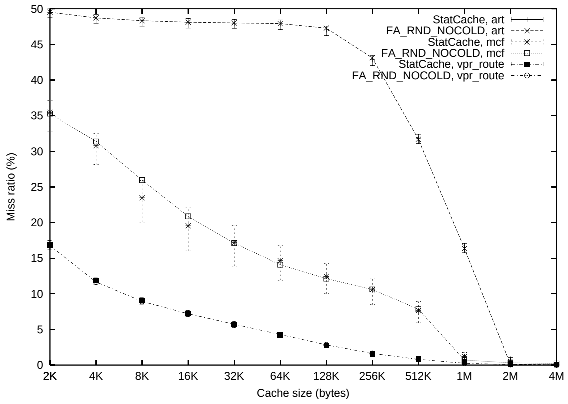
5 Implementation

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

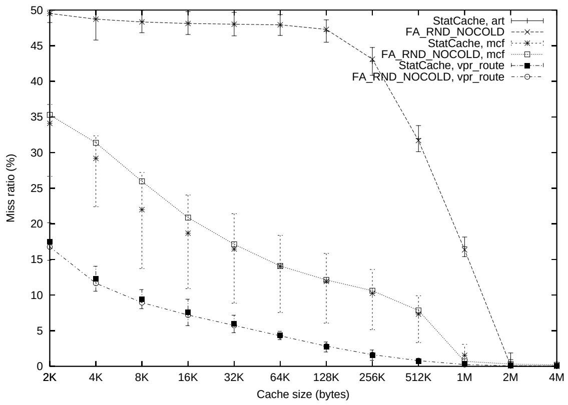


Figure 7: Example of error estimates for different sampling ratios. Note how the error bars grow as the sampling rate decreases. The error bars for mcf are very large for 0.00002 sampling rate, indicating that the sampling ratio is too low.

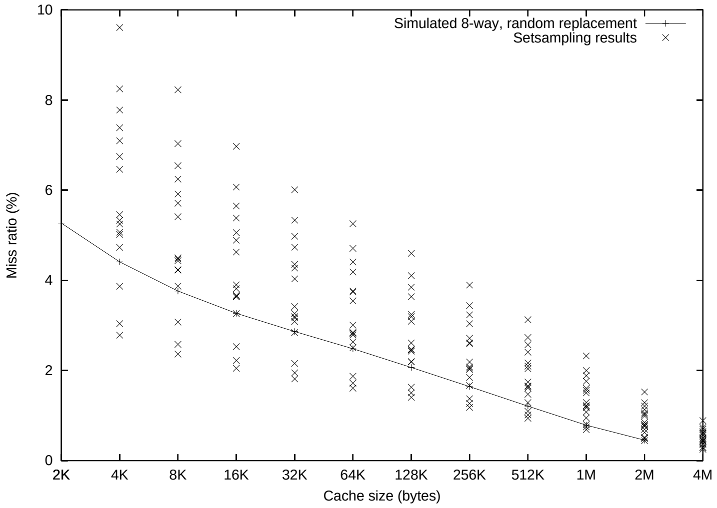


Figure 8: An example, bzip2_program, illustrating the large spread of estimated miss ratios when using set sampling. Crosses are different values from set sampling. This large spread is actually typical of set sampling and much reduces its use in comparative experiments.

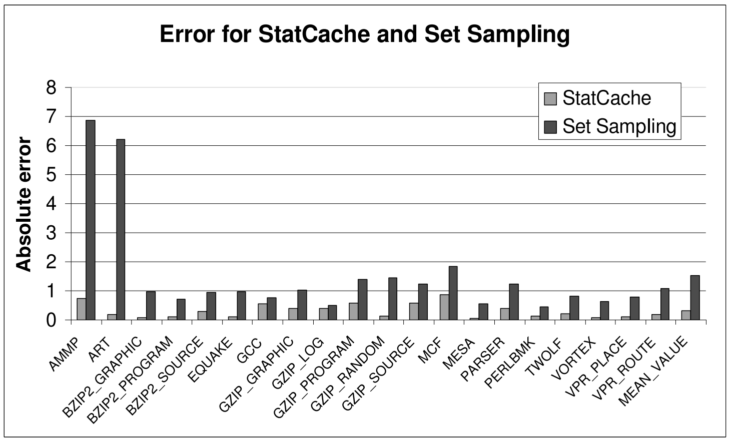


Figure 9: Deviation from the correct miss ratio for StatCache and set sampling. In the experiment, the sample rate for StatCache was set to 10^{-4} and every sixteenth set was used for the set sampling. The deviation of StatCache never exceeds 0.9 percent, while set sampling deviates almost 7 percent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

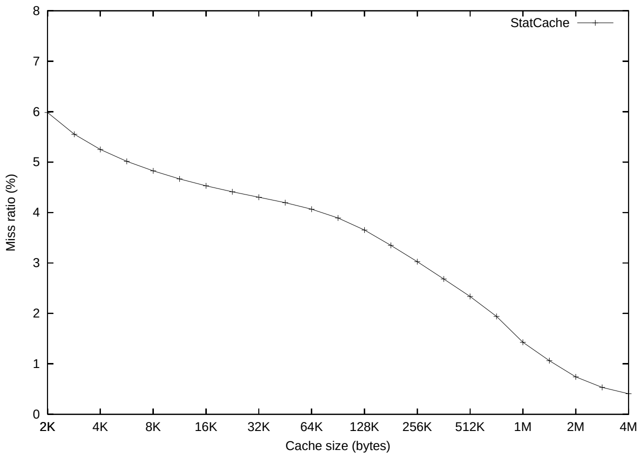


Figure 10: The output from the proof-of-concept implementation running 188.ammp with reference input. Observe the different shape compared to the corresponding curve in Figure 3 where we used a reduced input.

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.

Paper C

Low Overhead Spatial and Temporal Data Locality Analysis

Reprinted from

Technical report 2003-57
Department of Information Technology
Uppsala University
Uppsala, Sweden, November 2003.

Submitted for publication.

Low Overhead Spatial and Temporal Data Locality Analysis

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

November 25, 2003

Abstract

Performance is getting increasingly sensitive to cache behavior because of the growing gap between processor cycle time and memory latency. To improve performance, applications need to be optimized for data locality. Run-time analysis of spatial and temporal data locality can be used to facilitate this and should help both manual tuning and feedback-based compiler optimizations. Identifying cache behavior of individual data structures further enhances the optimization process. Current methods to perform such analysis include simulation combined with set sampling or time sampling, and hardware monitoring. Sampling often suffers from either poor accuracy or large run-time overhead, while hardware measurements have limited flexibility.

We present DLTune, a prototype tool that performs spatial and temporal data-locality analysis in run time. It measures both spatial and temporal locality for the entire application and individual data structures in a single run, and effectively exposes poor data locality based on miss ratio estimates of fully-associative caches. The tool is based on an elaborate and novel sampling technique that allows all information to be collected in a single run with an overall sampling rate as low as one memory reference in ten million and an average slowdown below five on large workloads.

1 Introduction

The use of caches and advanced memory hierarchies in modern computers assumes locality of data to perform well. Poor data locality causes the processor to stall frequently because many data accesses miss in the cache. Analysis of data locality helps in designing efficient software, both when used in a compiler framework to guide optimization decisions, and as a programming aid in the form of profiling tools. It is essential that such a tool is fast enough to allow for real-sized data sets and provides short turn-around times.

Data locality can be divided into spatial and temporal locality. A high degree of spatial locality means that data near recently accessed data are likely to be accessed again soon. Thus, the cache miss ratio for applications with much spatial locality should decrease with an increasing cache line size. Temporal locality instead means that recently accessed data are likely to be touched again soon.

To aid in a software optimization process, the profiling tool should distinguish between the two kinds of locality to prompt for suitable software program optimizations, such as loop-ordering decisions and tiling. Furthermore, the locality information for each data structure should be available to the programmer.

Current methods for implementing such performance tools include run-time cache simulation, and static program analysis. Run-time analysis include simulation, possibly combined with set sampling or time sampling, but can lead to large run-time overhead or limited accuracy. Traditional time sampling suffers from both cold start effects and the risk of selecting unrepresentative samples. Methods for static program analysis can be both accurate and fast but their use is limited to loop-nests, possibly with simple conditional clauses.

We present a prototype tool, DLTune, for profiling memory behavior in run-time. It produces generic spatial and temporal data-locality information for a whole application and its different data structures. The tool is based on a sampling technique that allow a very low overall sampling rate. In this paper we use a sampling rate as low as 10^{-7}. Our simple prototype implementation, based on code instrumentation, demonstrates a slowdown of 4.3 on average and we expect future implementations of the tool, based on hardware watchpoints, to cut that further.

In spite of its low overhead, enough information is collected from each run to allow for the post-process mathematical statistics presented in this paper to produce a plentiful of information from each run, such as the miss ratios of fully-associative caches of various sizes and line sizes. We also present measures of spatial and temporal locality that effectively exposes poor memory usage.

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: The data reuse distance is the number of intermediate memory references. 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.

2 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. The reuse distance of a memory reference that accesses a cache-line-sized piece of data is the number of memory references since the previous memory reference to the same piece of data. Figure 1 illustrates the concept. Cache line B is accessed in reference 1, and the next time in reference 6. Thus, the reuse distance for reference 6 is four. The reuse distance for the first reference to a cache line is undefined.

Note that the reuse distance defined above counts all intermediate memory references, in contrast to the stack distance[24] that only counts the number of different cache-line-sized pieces of data. For example, in Figure 1 the stack distance of the sixth reference is only 2. This is an important difference, because our reuse distance may be measured with a simple counter, while measuring the stack distance requires stack simulation.

2.1 Reuse Distance Histogram

The reuse distances of the memory references of an application can be viewed as a histogram. Figure 2 shows the reuse-distance histogram 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.

It is tempting to read out some quantitative cache performance information from such a diagram. It appears like a high population count for references with a low reuse distance would indicate good cache hit rate and vice versa. However, since the reuse distance only count the number of accesses, and not how many different cache lines are touched, we cannot estimate the miss ratio by just counting the fraction of memory references with a smaller reuse distance than the cache size, as is possible for the stack distance. This may look like a disadvantage compared to the stack distance. However, an important property of the reuse distance is that sparse sampling can easily be used to estimate its histogram shape, i.e., it is

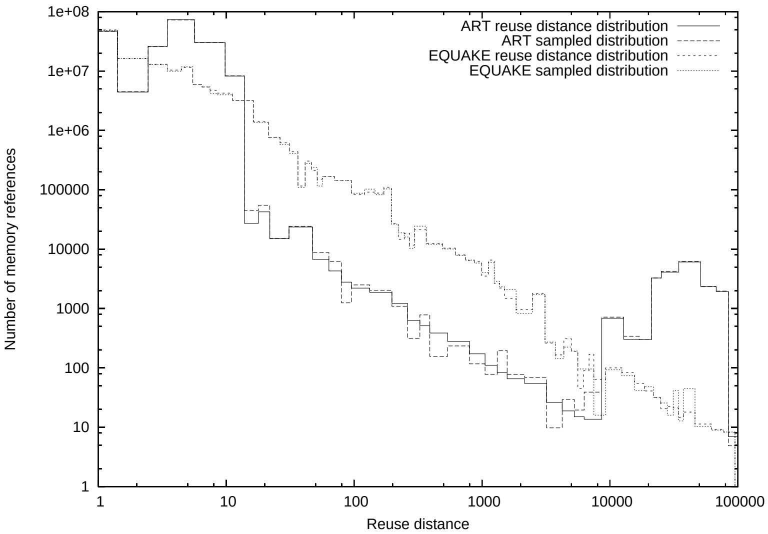


Figure 2: The diagram shows the reuse-distance distribution for the SPEC benchmarks art and equake. It shows distributions based on complete memory reference traces compared to sampling-estimated distributions.

enough to monitor the reuse distance for every Nth access. The sampling is done by randomly choosing one reference to sample. The cache-line-sized piece of data accessed by that reference is monitored until it is accessed again and its reuse distance is recorded. Implementing sparse sampling for stack distance would be substantially more complicated, since the history of all accesses must be examined between samples. Figure 2 shows two examples of reuse distance histograms and how they are approximated by sampling. Note that both axes have been made logarithmic in order to show the interesting shape of the curves.

Three basic mechanisms are needed to generate such a histogram through sparse sampling:

1. Sampling: A sampling mechanism selecting memory references to be monitored. This can for example be implemented with a timer interrupt, as long as every memory access is selected with the same probability.

2. Monitoring: A monitoring mechanism detecting new accesses to the address of the sampled references. Its implementation may rely on a watchpoint mechanism existing in many runtime systems or code instrumentation.

3. Access counter: A global memory reference counter incremented for each memory access. This functionality can often be provided by hardware counters.

2.2 Sampling Algorithm

The shape of the reuse distance histogram is insensitive to earlier memory behavior, since the first reference to a cache line has no reuse distance and is not added to the histogram.[1] The sampling method that DLTune implements takes advantage of this. While traditional time sampling must use long contiguous subtraces to warm cache models and avoid cold misses, DLTune can sample reuse distances in many short bursts. The large number of short bursts helps DLTune to better capture the cache behavior of applications with short periods of different cache behavior that are otherwise easy to miss. But even during the bursts, DLTune selects only sparse samples and not contiguous subtraces.

DLTune uses the following sampling parameters:

burst_length The number of samples taken within one burst.

sampling_rate The sampling probability within a burst.

burst_factor Determines the time between sample bursts. The time between bursts is $burst\_factor - 1$ times the length of a burst.

We use the following default settings in our experiments unless otherwise noted: $burst\_length = 60$, $sampling\_rate = 1/2000$, and $burst\_factor = 5000$. Each sampling burst is $burst\_length/sampling\_rate$ memory references long. The periods between the bursts are
$(burst\_length/sampling\_rate) \cdot (burst\_factor - 1)$ memory references long.

With default settings the tool samples 60 memory references during a burst of approximately $60/0.0005 = 120000$ memory references and the periods between bursts are $(60/(1/2000)) \cdot (5000 - 1) = 599880000$ memory references long. Thus, the tool takes 60 samples during a period of $12000 + 599880000 = 6 \cdot 10^8$ memory references with default settings. This gives an overall sampling rate of $1/10^7$.

2.3 Monitoring Memory

Cache lines are monitored using watchpoints. When the DLTune sampling mechanism has selected a cache line for monitoring, it sets a watchpoint to guard it and detect the next memory reference to the cache line. When the watchpoint detects the next memory reference, it calculates the reuse distance using the reference counter, and records the result. The watchpoint is removed from the cache line as soon as it has detected the first reuse.

[1] Cold references may affect the result because they are still counted by the reference counter, but we neglect that effect here.

To better handle applications with varying miss ratio, DLTune does not record every reuse distance in the same histogram, but creates a separate histogram for each sampling burst. This causes a problem, because every watchpoint do not detect reuse during the burst, but not until after the sampling burst is finished. In some cases, a watchpoint may live through several sampling bursts. Since a new histogram is created for each sampling burst, the reuse distance must be attributed to one of them. DLTune always attributes the reuse distance to the histogram corresponding to the burst during which the watchpoint was set. In practise, this has little impact on the shapes of the histograms because most reuse distances are very short, (See for example Figure 1. Note the log-scale.) and only a small fraction of the watchpoints do not trigger during the same burst as they were set.

Some access patterns cause the number of watchpoints to grow indefinitely. This could severly impair performance. Therefore, DLTune has an algorithm for removing very old watchpoints. First, a pair of very old watchpoints of similar age is identified, and one of them is removed. Then, the reuse distance of the remaining watchpoint will eventually be used as a substitute for the removed one.

2.4 Probabilistic Cache Miss Estimation

For a stack-distance histogram, the miss ratio of an arbitrary sized fully-associative cache can be estimated by determining the number of references with a stack-distance smaller than the cache size. The reuse distance defined in this paper is much easier to measure, but cannot be used to calculate miss ratios this way because it count all intermediate references, not just different ones. Instead, we use a probabilistic model of a cache. We assume a fully-associative cache, but assume random replacement policy instead of least recently used replacement.

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 *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 $r(i+1) + r(i+2) + \ldots + r(j-2) + r(j-1) = \sum_{i+1}^{j-1} r(i)$

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. This is a reasonable assumption for the relatively short sample bursts DLTune uses. 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 sampling-estimated reuse-distance histogram. Let h be the histogram. $h(0)$ 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[2]:

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

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

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

2.5 Average Application Miss Ratio

To calculate the overall miss ratio of an application, DLTune calculates the average of the miss ratio estimates of every sample burst. Although the error of the miss ratio estimate of each sample burst may be unacceptably large, the large number of such sample bursts tend to reduce the error, and we get an acceptable average result.

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.

The tool is also designed in such a way that it collects reuse-distance information for many cache line sizes in a single run. The tools ability to calculate miss ratio estimates for fully-associative caches of different cache sizes and cache-line sizes in a single run is the basis for further analysis of data locality, both spatial and temporal. The default settings for DLTune is to estimate miss ratios of caches with the cache line sizes 16 through 256.

3 The DLTune Prototype Implementation

The DLTune prototype implementation is based on code instrumentation. We use an assembly-code instrumentation tool [13] 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 a DLTune run-time system. The inline code snippet updates the reference counter and check if it is time to take a new sample. The snippet also uses a hash table to check if the load or store instruction references any monitored memory address. The snippet calls a handler if a new sample should be taken or a monitored memory address is referenced.

4 Experimental Evaluation

In an earlier paper we have evaluated a similar approach on very small data sets [10]. In this paper, we use trace driven simulation using the largest (ref) SPEC input sets on optimized code [3] to verify DLTune. The miss ratio estimates produced by DLTune are compared to miss ratios measured using a cache simulator. The simulated

[3]We used the Sun Forte Developer 7 C and Fortran compilers with the flags -O3 -xarch=v8plus -nolibmil -xregs=notool requires the two last flags.

cache is fully associative and uses random replacement. Address traces, generated by the same code instrumentation tool as DLTune uses, are directly feed to the cache simulator.

The native speed for running each application with this input is typically between 20 and 60 minutes. Since our DLTune-prototype only adds about five times overhead, it is still very feasible to run these large data sets. However, simulating fully associative caches with complete traces from the SPEC benchmarks with reference input is unfortunately not feasible, so we are forced to use shorter subtraces for the simulated caches. We skip the first 1 billion references and then simulate the next ten billion.

The miss ratios of the simulated caches and the estimates of DLTune are shown in Figures 3 and 4. The graphs show miss ratios for the cache line sizes 32 and 128 bytes and cache sizes from 8 K bytes to 4 M bytes. In contrast to the miss ratios of the simulated caches, the miss ratio estimates of DLTune is based on execution of the entire benchmarks with reference input, so we could expect some discrepancies for programs with varying miss ratios. [4] Most graphs are generated using the default sampling rate 10^{-7}, but for ART we doubled the default sampling rate to $2 \cdot 10^7$ because the trace is so short, only 38 billion references. The other benchmarks have between 85(mesa) and 340(mgrid) billion memory references. DLTune needs at least about 10000 samples to give reasonably good miss ratio estimates.

DLTune estimates the miss ratio well for most of the benchmarks. The tendencies are also well captured, which is important for understanding changes when comparing different program versions. The estimates for apsi and mesa may seem to have poor accuracy, but note that the absolute miss ratio values are small and that optimizing memory behavior is most important for applications with high miss ratios, like swim, art, applu and mgrid.

During the evaluation we noticed that subtraces are sometimes very unrepresentative of the entire application even for some of these floating point benchmarks, despite their simple repetitive algorithms. This affects our comparison and some of the errors are probably time-sampling errors connected to the simulated caches rather than to DLTune. Figure 5 illustrates the problem. Ammp has a very high miss ratio initially, but it soon drops to a low level. The curves in the figure show the miss ratio for the first one billion references and the miss ratio for the next ten billion references, i.e. memory reference $1 \cdot 10^9$ to $11 \cdot 10^9$. Clearly, hitting the right miss ratio with a subtrace is often a matter of luck, even if it is one billion

[4]The subtrace for mesa turned out to be very unrepresentative for the overall miss ratio, so we run DLTune only on the subtrace, and increased the sampling rate to get approximately the same amount of samples as for the full trace. The average error when we run DLTune on the entire mesa benchmark and compared to the simulated cache was still less than 0.5 percent units.

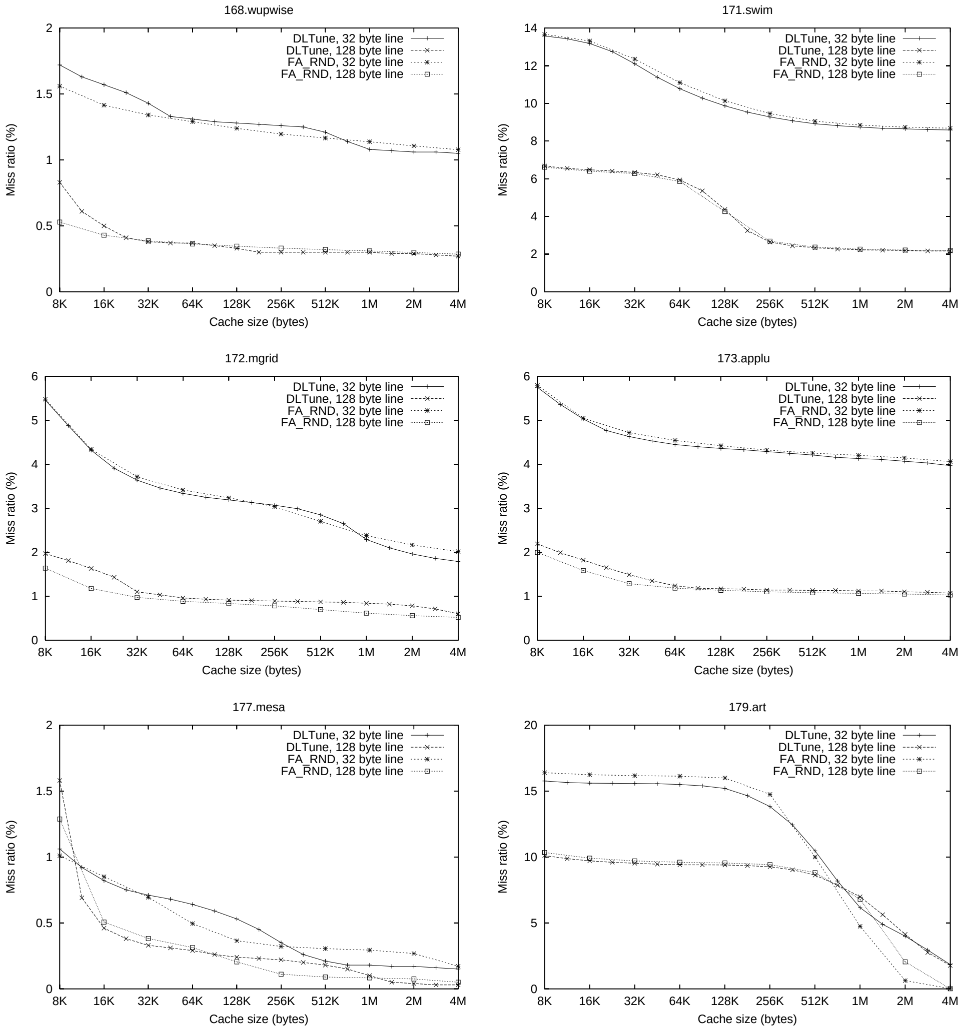


Figure 3: Comparison between DLTune and simulated fully associative caches (FA_RND). The miss ratio estimates of DLTune are close to those of the simulated fully in most cases. 177.mesa may seem to have poor accuracy, but note that the absolute miss ratios and errors are low, mostly less than 0.3 percent units. The sampling rate is 10^{-7} for all experiments except art and mesa. The reasons are explained in Section 4. More benchmarks are shown in Figure 4

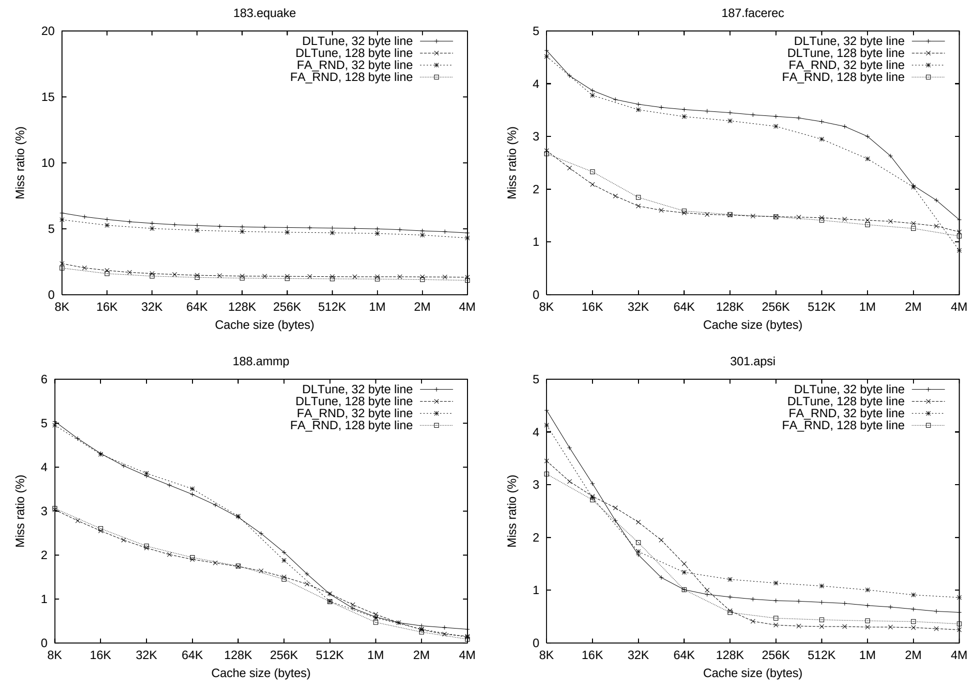


Figure 4: Comparison between DLTune and simulated fully associative caches (FA_RND). Continued from previous page.

references long. In the ammp case the miss ratio stays low after the short initial period with very high miss ratio, so the longer trace should still be representative of the rest of the application.

4.1 Impact of Replacement Policy

DLTune models caches with random replacement to study memory behavior, while other analysis methods are based on LRU replacement[25][5] [29]. The difference is not important as long as they both lead to the same optimization decisions. We have simulated caches with both replacement policies to examine this and found that the differences are limited for most applications, but small differences exist. Figure 6 shows the miss ratios of the four benchmarks ammp, art, bzip2_program and equake for fully associative caches with random and LRU replacement. Note that the tendencies are the same while the absolute values differ slightly. The absolute values are different from those in Figures 3 and 4 because we used complete simulation with reduced input sets[16] in this study.

For small caches, LRU replacement gives a smaller miss ratio than random re-

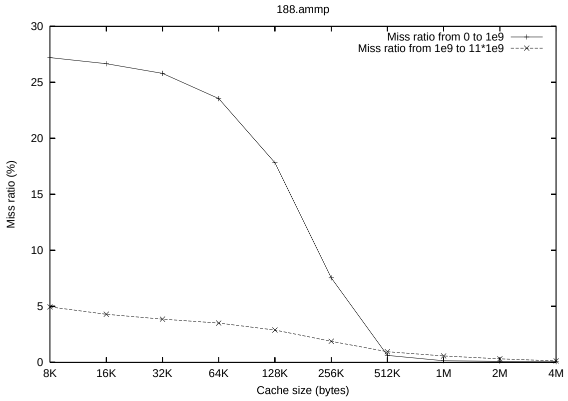


Figure 5: The miss ratio of some applications can be hard to estimate with a short subtrace. The diagram shows the miss ratio of 188.ammp as an example of this. The two curves show the miss ratio for the first billion memory references, and the next ten billion references.

placement for bzip2_program and equake but the difference vanishes as the cache size grows. This is common for many applications. For art and ammp, LRU replacement policy tend to give larger miss ratios than random replacement policy for caches between 64K byte and 1M byte. This is typical behavior of applications that traverses large amounts of data without reusing it for a long time. Compare with Figure 2, where the reuse-distance histogram of art has a bump for large reuse distances.

Another motivation why random-replacement fully-associative caches should be useful for data-locality analysis is the following: Consider a direct mapped cache. If the access pattern of an application causes all memory references to hash uniformly across all 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 should therefore have a miss ratio similar to that of a direct-mapped cache. A similar argument holds for set-associative caches with random replacement. Thus, miss ratio estimates of DLTune should give appropriate measures of locality for many realistic caches.

5 Data Locality Optimization

Temporal and spatial data locality affect different aspects of program optimization. They could still affect each other, because poor spatial locality causes misses that

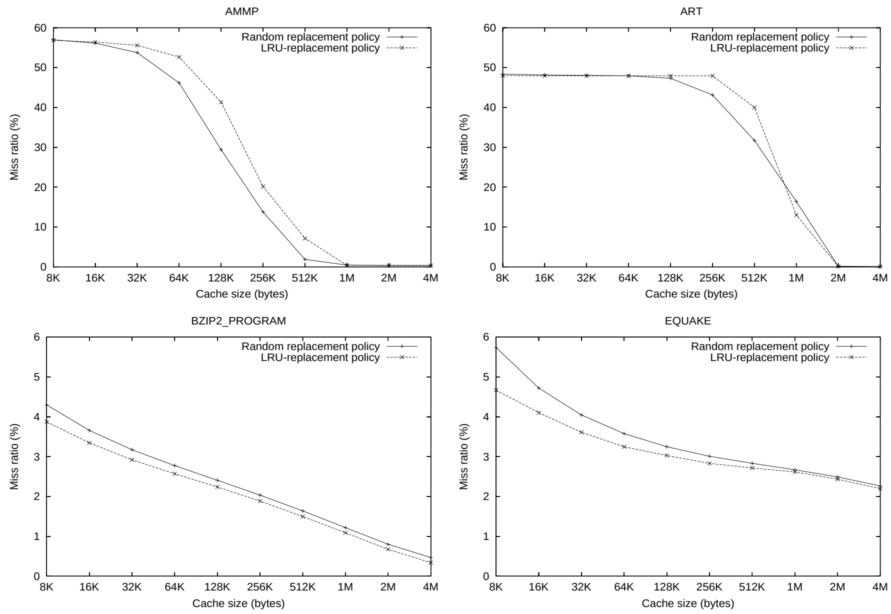


Figure 6: Examples of the difference between LRU (FA_LRU) and random (FA_RND) replacement policies for fully associative caches. The figures show the miss ratios of simulated fully associative caches for the benchmarks art, gzip_program, mcf, vpr_place, and bzip2_graphic. The graphs show that the two replacement policies usually give similar behavior. The miss ratios in these graphs may differ from other graphs in the paper because they are produced with smaller input sets.

reduces temporal locality and vice versa. We exemplify the two types of locality before we give quantifications of them.

5.1 Temporal Locality Optimization

We demonstrate temporal tuning with a standard blocked matrix multiplication. Figure 7 shows the code. If the block sizes b varies, the miss ratio should change if the working set size is close to the cache size. Figure 8 shows what happen when b is varied between 20 and 100. The graphs show the miss ratio estimates of DLTune for 16 and 64 K bytes caches with the line sizes 16 and 32 bytes respectively, compared to the first level data caches of Ultra SPARC II and III that have the corresponding sizes. The Ultra SPARC II cache is subblocked, but the basic fill unit is 16 bytes. Note that DLTune captures the changes accurately despite an offset. A probable reason for the offset is that DLTune models fully-associative caches,

```
do jj = 1, N, b
   do kk = 1, N, b
      do i = 1, N
         do j = jj, jj + b - 1
            r = 0.0
            do k = kk, kk + b - 1
               r = r + y(i,k) * z(k,j)
            end do
            x(i,j) = r
         end do
      end do
   end do
end do
```

Figure 7: Blocked Matrix Multiplication

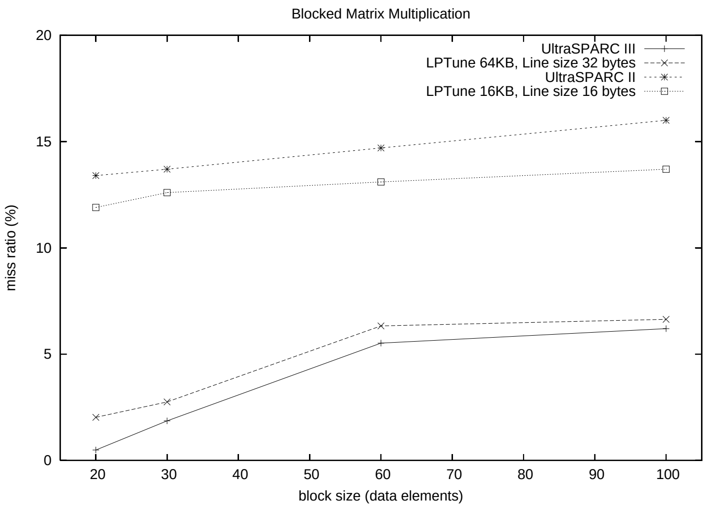


Figure 8: Miss ratio of blocked matrix multiplication for different block sizes. Note that the estimates by DLTune reacts to changes in the block size just as the real caches on UltraSPARC II and III, despite some offset.

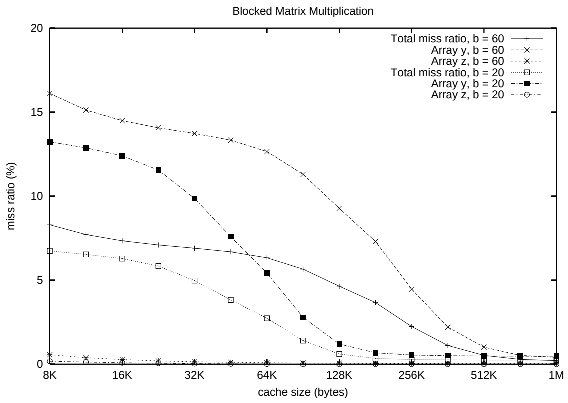


Figure 9: DLTune allows individual arrays to be analyzed. The graphs show miss ratios for the arrays y and z as well as total miss ratio.

whereas the hardware caches are direct-mapped and 4-way set-associative respectively. The offset is largest for the 16 K byte direct mapped-cache, which indicates that there probably are some conflict misses as well as capacity misses. For the 64 K byte cache, the hardware actually gives a lower miss ratio. There are applications that benefit from low associativity, but it could also be interference form other processes, or a DLTune approximation or sampling error. However, DLTune is designed primarily to capture data locality and not exactly model cache behavior. The important thing is that DLTune captures trends for proper optimization decisions.

The DLTune approach can also show miss ratios for references to individual data structures. Figure 9 illustrates this. The array y has the largest miss ratio, because it is referenced in column major order. For more advanced codes, this should turn out useful to track data structures with unfavorable access patterns.

5.2 Spatial Locality

The benchmark swim performs iterative updates of a number of two-dimensional arrays. In each iteration, the program executes three two-level loop-nests, Figure 10 shows one of them. The arrays are too large for much of the data to remain in the cache until the next iteration. Especially in small first level caches, almost only spatial locality can be expected. Assembly code inspection showed that the program uses 32-bit instructions to read data. The miss ratio for caches with 16 and 32 byte lines should therefore be approximately $1/4$ and $1/8$ respectively.

Figure 11 shows the miss ratio estimates of 179.art and 171.swim produced by DLTune as a function of the cache size. Swim behaves as expected. The cache

```
 DO 100 J=1,N
  DO 100 I=1,M
   CU(I+1,J) = .5D0*(P(I+1,J)+P(I,J))*U(I+1,J)
   CV(I,J+1) = .5D0*(P(I,J+1)+P(I,J))*V(I,J+1)
   Z(I+1,J+1) = (FSDX*(V(I+1,J+1)-V(I,J+1))-
                FSDY*(U(I+1,J+1)-
                U(I+1,J)))/(P(I,J)+P(I+1,J)+
                P(I+1,J+1)+P(I,J+1))
   H(I,J) = P(I,J)+.25D0*(U(I+1,J)*U(I+1,J)+
            U(I,J)*U(I,J)+V(I,J+1)*V(I,J+1)+
            V(I,J)*V(I,J))
100 CONTINUE
```

Figure 10: A loop nest from swim. The program spend most of its time in simple loops like this.

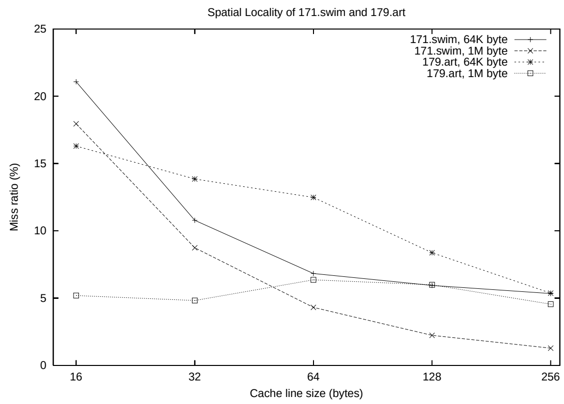


Figure 11: Spatial locality of 179.art and 171.swim. The graphs show the miss ratio of the two benchmarks estimated ny DLTune as a function of the line size for the cache sizes 64K byte and 1M byte. Art has some spatial locality for the 64K byte cache but none for the 1M byte cache. Swim shows a very high degree of spatial locality for both cache sizes.

miss ratio approximately halves when the cache size doubles. The exception is the right part of the 64K cache, were the miss ratios are very similar for the line sizes 64, 128 and 256. The reason is that the cache is so small that data is evicted before the program can take advantage of all the potential spatial locality. Spatial locality is thus limited to 64 bytes for the 64K byte cache. In contrast, the miss ratio of the 1M byte cache continues to decrease for cache line sizes up to 256 bytes. The absolute values are somewhat lower than expected, but this is probably because there is limited temporal locality as well. The curve for the 1M byte cache is also lower than the 64K byte curve, which indicates some temporal locality for larger caches.

Art behaves differently. The miss ratio of the 64 K byte cache decreases towards larger cache line size, while the miss ratio of the 1 M byte cache is approximately constant. For the largest cache line size, 256 bytes, the miss ratio for both cache sizes are similar. We conclude that the small cache is too small to fully take advantage of the temporal data-locality in the benchmark, while the large cache is. Further, there is very little spatial locality to take advantage of in the large cache. Thus, spatial locality depends on cache size, and to get a complete view of spatial locality, we need to study it for different cache sizes.

6 Quantifying Spatial Locality

Comparing the miss ratio of caches of the same size, but with different cache line sizes, allows quantification of spatial locality. If the miss ratio drops by 50 percent when the line size is doubled, the spatial locality is good. Likewise, quadrupling the line size reduces the miss ratio to a fourth. Let B_{small} be the smallest cache line size that DLTune estimates miss ratios for, and B_{actual} be the line size of the hardware our optimizations targets. R_{small} is the miss ratio for cache line size B_{small}, and R_{actual} is the miss ratio for cache line size B_{actual}. For a given cache size, define the spatial use:

$$Spatial\ Use = \frac{1 - R_{actual}/R_{small}}{1 - B_{small}B_{actual}}$$

The Spatial Use will be one if the miss ratio is halved when the cache line doubles, and zero if the miss ratio is unchanged. Table 1 shows the spatial use for the ten SPEC benchmarks we investigate. Two benchmarks, art and ammp, stands out with very poor data locality. This indicates that data is accessed in an unfavorable way in the two applications. For example ammp initializes large arrays of C-structures field by field instead of all fields for one array-index at a time, which causes very poor spatial locality.

Benchmark	Spatial Use	Benchmark	Spatial Use
168.wupwise	1.07	179.art	-0.30
171.swim	1.01	183.equake	0.98
172.mgrid	0.93	187.facerec	0.80
173.applu	0.97	188.ammp	-0.02
177.mesa	0.81	301.apsi	1.01

Table 1: Spatial Use for ten SPEC benchmarks. Most of the applications show very good spatial locality, while two, art and ammp, stands out with poor locality. The cache is 1M byte with 64 byte lines.

Benchmark	Temporal Use	Benchmark	Temporal Use
168.wupwise	37	179.art	13
171.swim	5.6	183.equake	10
172.mgrid	22	187.facerec	20
173.applu	12	188.ammp	115
177.mesa	131	301.apsi	57

Table 2: Temporal Use for ten SPEC benchmarks in a 1M byte cache with 64 byte lines.

7 Quantifying Temporal Locality

An intuitive way to think about temporal locality is how many times a program uses a variable before it is evicted from the cache. It is not possible to just count the average number of accesses to a large cache line, because there may also be spatial locality. It is better to have a measure that only includes actual temporal reuse of the same data. The easy way is to use the miss ratio of the smallest cache line, 16, that DLTune estimates and invert it. This gives a number that ranges from one to infinity. A value close to $16/word\_size$ means that data is referenced only once during its tenure in the cache, and large values indicate good temporal locality.

Swim has poor temporal locality and benefit from blocking[25]. It references memory by 32-bit instructions so the lowest possible temporal use value is $16/4 = 4$. Table 2 shows that the temporal use is 5.6 in a 1M byte cache with 64 byte lines. There is almost no temporal use, despite the miss ratio below five percent (Fig. 11). Art has a miss ratio above six percent, but still has the temporal use thirteen. This correlates well with the poor spatial use of art. Thus, we need to look at small cache line sizes to study temporal behavior. Just measuring the miss ratios of caches with actual line sizes does not expose all aspects of data locality.

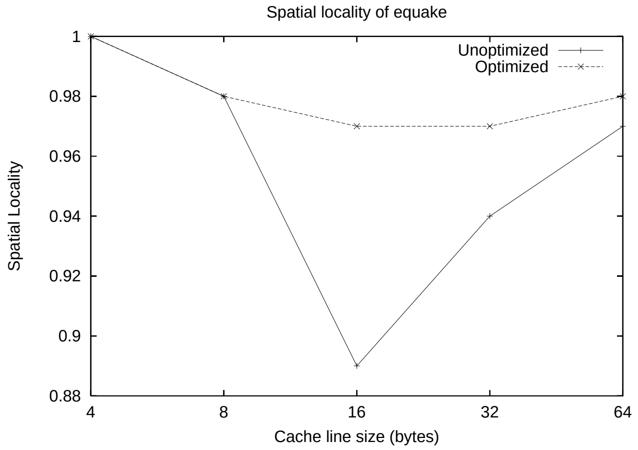


Figure 12: Spatial locality of SPEC 183.equake. The graphs show the Spatial use defined in Section 6 as a function of cache line size for the original (unoptimized) and optimized code. The spatial locality of the optimized version is close to one for all line sizes, while the spatial locality for the unoptimized version drops for 16 byte lines.

8 Spatial Locality of SPEC 183.equake

In a previous paper [2] we have shown that the spatial locality of the equake benchmark can be optimized. This may seem odd because Table 1 says that the spatial use of equake is good. However, it turned out that even 16, the default smallest cache line size of DLTune, is too large to capture the spatial locality problem in equake, it is not as sever as in ammp for example. Therefore, we run DLTune on equake with the minimum cache line set to 4 bytes instead.

In the original version of code, the memory is allocated in small chunks (6 bytes large) with unused wholes between them. We optimized the code and found that the benchmark run about 10 percent faster compiled as 32-bit program and 43 percent faster if compiled as 64-bit. For this study we used a minimum cache line size of 4 bytes instead of the default line size 16 bytes. Figure 12 shows the spatial locality of both the original program and an optimized version with an improved memory allocator. Note that the spatial use is close to one for all cache line sizes for the optimized version, while the original program has a sharp dip for 16 byte lines. The graphs show the spatial locality for the 32-bit version of the program only, because our prototype tool can only handle 32-bit programs. This study shows that we can find the same spatial locality problems with DLTune as with our earlier profiling tool [2] based on functional cache simulation.

Benchmark	Average watchpoints	Percent watchpoints removed by aging algorithm
168.wupwise	2.0	1.5
171.swim	6.8	0
172.mgrid	3.0	0
173.applu	3.7	0
177.mesa	2.4	0.01
179.art	6.9	0.13
183.equake	4.0	0
187.facerec	3.0	0.04
188.ammp	3.5	0.10
301.apsi	1.7	1.3

Table 3: The average number of watchpoints set at a time during sample bursts and the fraction removed by the aging algorithm. Note that only a handful of watchpoints are needed on average for most of the benchmarks.

9 Performance

The prototype DLTune has an average slowdown of 4.3 on the tested benchmarks with their largest (*ref*) input. The slowdown is largely determined by the inserted code snippets next to every load and store instruction. An implementation using a different form of watchpoint mechanism, for example hardware watchpoints, could prove much faster. Table 3 shows how many watchpoints that are used at a time on average during the sample bursts, and how many watchpoints that are removed because of high age, as described in section 2.3. It is clear that the tool only needs a limited number of watchpoints at a time despite only a small fraction must be removed because of old age. This, together with the fact that a very low sampling rate suffice, indicates that an implementation with limited hardware support to replace the code instrumentation could prove very efficient.

10 Related Work

The study of data locality is the basis for understanding many aspects of application performance. There are several approaches to this, the most common is to estimate the miss ratio of caches of various sizes. The major approaches are compile-time and run-time analysis.

Compile time analysis are represented by for example[28] [5]. They try to estimate cache miss ratios by analyzing the code and try to figure out when cache misses should occur. Their major advantage is that they don't require the program to be executed, and can potentially be parameterized in terms of workloads etc. Their drawback is their limitations to relatively well-structured codes, and are mostly applicable to numeric algorithms.

Run-time analysis include cache simulation and hardware counter recording, both possibly combined with sampling. Cache simulators may be driven by instrumented code[20] [11][17][9][18] [22][23], on source code [14] or machine code levels, or the cache simulator incorporated in a full system simulator[21][19]. Their major limitation is their large slowdown, especially full system simulators.

Hardware counters are available on most modern computers. Events that can be counted include L1 and L2 cache misses, coherence misses and number of stall cycles. Examples of use include 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 [26] and locate problem areas [4]. Their limitations are mainly that only architectural parameters realized on the hardware may be studied, and that it can be hard to capture entities not directly present in the hardware, such as spatial locality.

Sampling techniques include set sampling and time sampling. 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 [30][15] [8][12]. 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 [15] [8]. It generally suffers from poor accuracy and can only be used as a rough estimate. Trap-driven trace generation has also been suggested [27]. It can trace unmodified code, but requires OS modification.

Other approaches to describe and quantify memory behavior include the concept of data streams or strides. Information about data streams can be used to guide prefetching[6] [7] and help choose between optimizations such as tiling, prefetching and padding[25].

11 Conclusions

This paper presents DLTune – an efficient tool for analyzing data locality and to distill information which is useful for program optimization. DLTune relies on a new algorithm based on very sparse samples. The results are accurate enough to be useful in a tuning process, and do not suffer much from the risk that traditional time sampling suffers from, namely picking unrepresentative samples.

DLTune can guide a programmer, or compiler, to different approaches for optimizations, for example evaluate the effect of program transformations such as blocking and loop interchanges. As a guidance, the tool presents the miss ratios for caches of different sizes and cache line sizes. We also define measures of spatial and temporal locality, that are easily calculated from the data produced by DLTune, and effectively exposes poor data locality.

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. SIP: Performance Tuning through Source Code Interdependence. In *Proceedings of the 8th International Euro-Par Conference (Euro-Par 2002)*, pages 177–186, Paderborn, Germany, August 2002.

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

[6] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 191–202, 2001.

[7] T. M. Chilimbi. Dynamic hot data stream prefetching for general-purpose programs. In *PLDI*, 2002.

[8] 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.

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

[10] E.Berg and E.Hagersten. StatCache: A probabilistic approach to efficient and accurate data locality analysis. *Technical report 2003-57, Department of information technology, Uppsala University, Sweden*, 2003.

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

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] T. Mohan, B. R. de Supinski, S. A. McKee, F. Mueller, A. Yoo, and M. Schultz. Identifying and exploiting spatial regularity in data memory references. In *Proceedings of Supercomputing*, 2003.

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

[27] 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.

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

[29] 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.

[30] 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 licentiate theses from the Department of Information Technology

2003-003 Inger Boivie: *Usability and Users' Health Issues in Systems Development*

2003-004 Malin Ljungberg: *Handling of Curvilinear Coordinates in a PDE Solver Framework*

2003-005 Mats Ekman: *Urban Water Management - Modelling, Simulation and Control of the Activated Sludge Process*

2003-006 Tomas Olsson: *Bootstrapping and Decentralizing Recommender Systems*

2003-007 Maria Karlsson: *Market Based Programming and Resource Allocation*

2003-008 Zoran Radovic: *Efficient Synchronization and Coherence for Nonuniform Communication Architectures*

2003-009 Martin Karlsson: *Cache Memory Design Trade-offs for Current and Emerging Workloads*

2003-010 Dan Wallin: *Exploiting Data Locality in Adaptive Architectures*

2003-011 Tobias Amnell: *Code Synthesis for Timed Automata*

2003-012 Olivier Amoignon: *Adjoint-Based Aerodynamic Shape Optimization*

2003-013 Stina Nylander: *The Ubiquitous Interactor - Mobile Services with Multiple User Interfaces*

2003-014 Kajsa Ljungberg: *Numerical Methods for Mapping of Multiple QTL*

2003-015 Erik Berg: *Methods for Run Time Analysis of Data Locality*

UPPSALA
UNIVERSITET

Department of Information Technology, Uppsala University, Sweden