Fast Data-Locality Profiling of Native Execution

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

Performance tools based on hardware counters can efficiently profile the cache behavior of an application and help software developers improve its cache utilization. Simulator-based tools can potentially provide more insights and flexibility and model many different cache configurations, but have the drawback of large run-time overhead.

We present StatCache, a performance tool based on a statistical cache model. It has a small run-time overhead while providing much of the flexibility of simulator-based tools. A monitor process running in the background collects sparse memory access statistics about the analyzed application running natively on a host computer. Generic locality information is derived and presented in a code-centric and/or data-centric view.

We evaluate the accuracy and performance of the tool using ten SPEC CPU2000 benchmarks. We also exemplify how the flexibility of the tool can be used to better understand the characteristics of cache-related performance problems.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Measurement, Performance

Keywords

cache behavior, profiling tool

1. INTRODUCTION

An ideal profiling tool should have low run-time overhead and high accuracy, it should be easy to use and flexible, and it should provide the user with intuitive and easily-interpreted information. Low run-time overhead and high accuracy are both needed to efficiently locate performance bottlenecks with short turn-around time, and the ease-of-use requirement excludes methods which need cumbersome experimental setups or special compilation procedures. The tool should also be able to profile any application in its normal development, test and even production environment. Flexibility means that the profile should be applicable to a variety of hardware configurations and application input-data sets, and the tool should detect performance issues that do not affect performance on the experiment platform but may affect performance on other hardware. To produce easily-interpreted profiles, the tool should present code-centric and data-centric information.

It is unfortunately hard to combine all the requirements above in a single tool. For example tools based on hardware counters usually have a very low run-time overhead, but their flexibility is limited because hardware parameters like cache and TLB sizes are defined by the host computer [1, 12, 16, 30]. Simulators on the other hand are very flexible but are usually slow [13, 21, 14, 23, 26]. At worst, they may force the use of reduced data sets or otherwise unrepresentative experiment setups that give misleading results. Furthermore, simulators often need a special compiler or require the application to be installed on a virtual simulated machine, which leads to complex usage.

We propose tools based on statistical models as an alternative to hardware-counter-based and simulation-based approaches. Such tools have the potential to meet all of the demands for flexibility, speed and ease-of-use. They can be parameterized since they do not rely on the host hardware configuration, they are potentially very fast because they can be based on sparse sampling and they can also provide a simple and efficient user environment. While there are many performance aspects that should be handled by a generic tool, this paper focus on data cache behavior. This paper presents StatCache, a proof-of-concept implementation of a fast cache profiling tool based on a statistical modeling technique.

The capabilities of StatCache include locating source code lines that cause poor cache utilization and analyzing data interdependencies between source code lines. StatCache can also generate working-set graphs and the paper describes measures of spatial locality derived from the data StatCache produces. StatCache can profile unmodified single-threaded applications running natively on the host, including the cache effects of library code. The run-time overhead is on average less than 40% for the ten benchmarks in the evaluation. All benchmarks are from the SPEC CPU2000 [33] suite and in-

SIGMETRICS'05, June 6–10, 2005, Banff, Alberta, Canada.

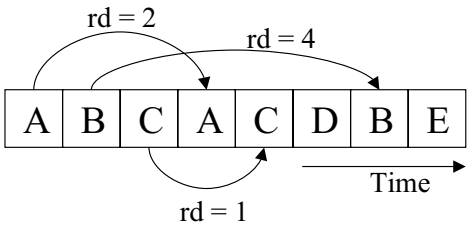

Figure 1: This figure illustrates the reuse distance concept. Assume that the letters A, B, C, A, ... in the boxes represent cache-line-sized pieces of memory accessed in that order by a target application. The reuse distance, rd, is the number of intervening accesses to other memory locations. For example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

clude complex applications like *gcc* and *gzip*. StatCache runs entirely in user space and relies on operating-system and hardware support available on most platforms. The current implementation runs on the SPARC/Solaris platform. The paper describes several issues that had to be solved to get a working tool, for example how to select samples, and it also describes how to map the statistical information to individual machine instructions and source code lines.

The next two sections will give a short introduction to the previously published theoretical background and present the implementation in a top-down manner. Sections 4 and 5 describes the validation of StatCache and discuss how to use StatCache for data locality optimizations. Related work is presented in Section 6 and, finally, in Section 7 we conclude.

2. THEORETICAL BACKGROUND

StatCache uses a statistical model for estimating the miss ratio of fully associative caches. This section gives a brief presentation of the theoretical background of StatCache[4]. We use the term *target application* to denote the analyzed application.

2.1 Reuse Distance

The reuse distance plays a central rôle in StatCache. We define the reuse distance as the number of memory accesses between two accesses to the same cache-line-sized piece of memory. Figure 1 illustrates the concept, where the processor accesses cache line A, then accesses two other cache lines (B and C in the figure), and finally the processor accesses cache line A again. Thus, the reuse distance of the second access to A is two. Please note that this definition of the reuse distance counts all intermediate memory accesses, contrary to the stack distance that only counts unique memory accesses [28]. This is an important difference, the reuse distance defined above is much easier to measure than the stack distance and enables an efficient implementation.

2.2 The Statistical Model

The reuse distances of an application largely determine its cache behavior. These reuse distances are hard to interpret as is, but can be transformed into cache-miss ratios using a statistical model. The model gives the miss ratio of a fully associative cache of arbitrary size with random replacement. Please see our previous publication [4] for a detailed description of the model and a simulation-based validation.

Assume that we know the reuse distance of a representative subset of all memory accesses. This subset can be selected by sampling. Then sort the reuse distances of all the memory accesses of the subset into buckets, h_i. Let h_0 be the number of memory accesses with reuse distance zero, h_1 the number of memory accesses with reuse distance one, and so on. Then solve the equation

$$R \cdot N = h_1 f(R) + h_2 f(2R) + h_3 f(3R) + \ldots \qquad (1)$$

for R, which is the miss ratio. N is the total number of samples, i.e. $N = h_0 + h_1 + h_2 + \ldots$, and $f(n)$ is a function that gives the probability that a cache line has been evicted from the cache if we know that it was in the cache n cache misses ago. The function f is

$$f(n) = 1 - (1 - 1/L)^n \qquad (2)$$

where L is the number of cache lines in the cache. The cache size is L times the cache line size.

Equation 1 is only valid if the miss ratio is approximately constant during the execution, which is not always the case. To handle this, StatCache divides the target-application run into several short periods, where each period is short enough for the miss ratio to be approximately constant during that period. Such a period of the execution is called a *sampling window*. The basic theory is applied to each sampling window. Equation 1 gives an estimated miss ratio for each sampling window and the overall miss ratio of the application is simply the arithmetic mean of the miss ratio of every sampling window.

3. IMPLEMENTATION

This is a top-down description of our profiling tool StatCache, starting with the user interface and working our way toward important implementation details.

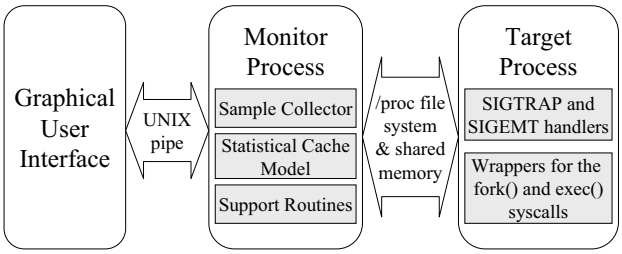

Figure 2: The figure shows the graphical user interface, the monitor process and a target process. The user interface contains only code for presenting results while the monitor process handles all sample-data collection, the statistical cache model and other support code for example for reading debugger information. The trap handlers and syscall wrappers are injected in the target process at load time using the preload support in Solaris. The processes communicate through UNIX pipes, shared memory and the `/proc` file system.

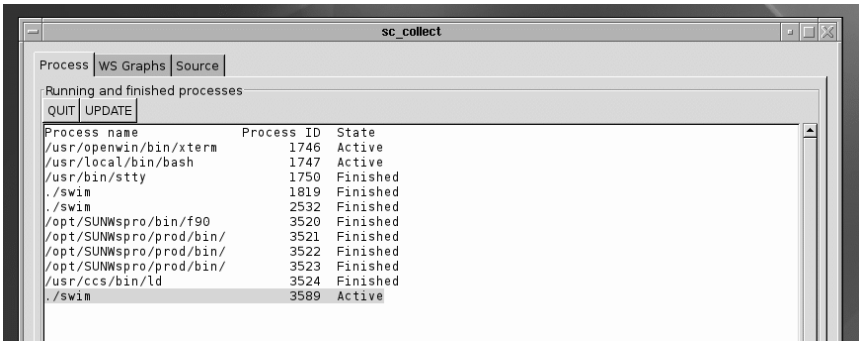

Figure 3: The process view of the user interface shows a list of all executables monitored by StatCache, their process ID and state, active or finished. New processes are added to the list when they are started and a user can select any process, active or not, and then switch to one of the other views to display detailed cache miss information about the selected process.

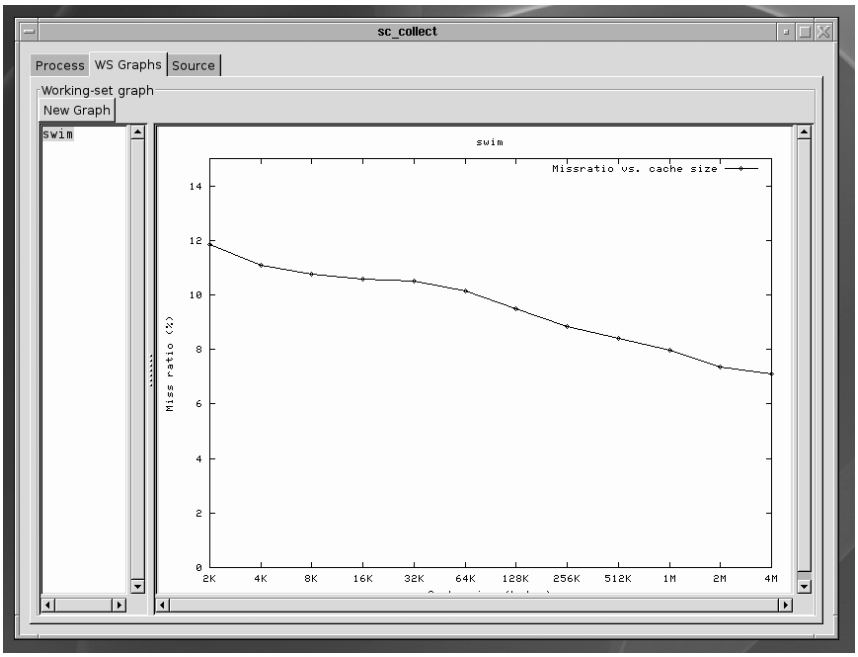

Figure 4: The working-set view of the graphical user interface. The tool can present graphs that show the miss ratio as a function of cache size for any target process. This is for example useful for analyzing the impact of different input data sets on cache utilization and for tuning blocking factors. This picture shows the working-set graph for SPEC *swim*. The miss ratio is apparently not very sensitive to cache size, thus, we would not gain much performance by moving the application to a computer with larger caches.

3.1 Overview

Figure 2 gives an overview of the tool, which is composed of two parts, a graphical user interface for presenting profiling results and a monitor process that handles sampling and the statistical calculations. The figure also shows a target process and how the different processes communicate. This design enables the implementation of a tool that is interactive in the sense that profiling information is available at run-time and is continously updated during execution.

A single command at a shell prompt with the target application as argument starts the profiler. While the tool keeps track of new subprocesses, an easy way to use the tool interactively is to first start an `xterm(1)` (X terminal emulator) as the first process from the command line, and then start the actual target application in the new xterm window. Thus, the same target application can be rerun several times using this approach without restarting the tool. This enables easy comparison of the profiles of different runs for example to study the impact of different input data sets on cache behavior.

The StatCache implementation runs on UltraSPARCIII-based computers and can be used to profile unmodified binary code. The entire tool runs as an unprivileged program under Solaris because no special drivers are needed and no part of the tool must run as a privileged user. StatCache is written in C and consists today of about 11,000 lines of code. It supports the stabs debugging format generated by the Sun C and F90 compilers.

3.2 The User Interface

The user interface displays profiling information for all target processes. It has three views, and a user can switch between them using the tabs at the top. The first view is a list of running and finished target processes, identified by name and process id, as seen in Figure 3. In this example we have first started an xterm window and then run SPEC *swim* twice, recompiled it with the Sun F90 compiler and linker, and restarted *swim* again. StatCache collects performance data independently for each of these processes. After selecting one of the processes, a user can switch to one of the other two views to study the available profiling information for that process. The second view presents working

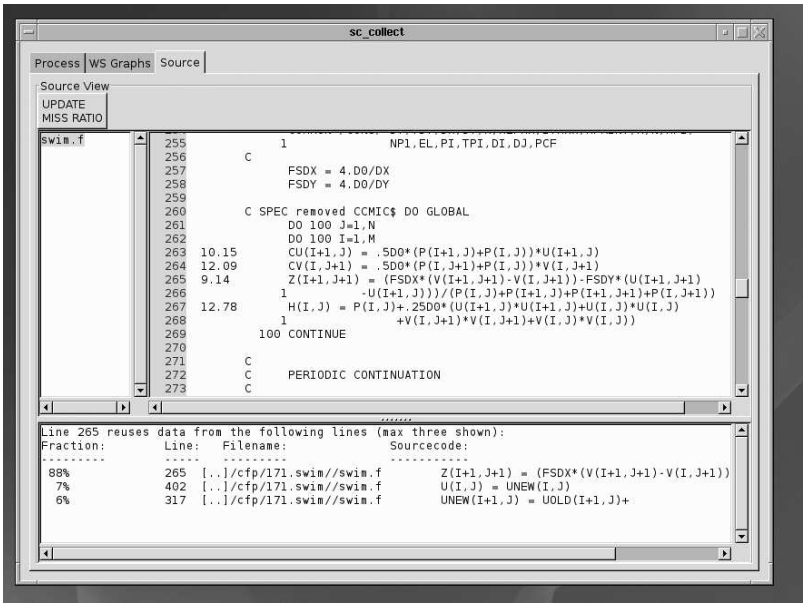

Figure 5: The source view shows code-centric profiling information. The view has three panes, the upper left shows the source file names (only *swim.f* in this example), the upper right pane shows source code annotated with the miss ratio per code line, and the bottom pane shows detailed data dependency information. The bottom view shows in this example that most (88%) data reuse for line 265 is self reuse, that is, data is previously accessed by the same line.

set graphs for the selected target process, i.e., miss ratio as a function of cache size. The screen shot in Figure 4 shows such a graph for *swim*. The third view shows the source code of the application annotated with cache miss rate information. Figure 5 shows the source code for *swim*, where lines 263, 264, 265 and 267 are annotated with cache miss ratio numbers next to the line numbers. The bottom half of this view contain data reuse information.

3.3 The Monitor Process

The monitor process is the core of StatCache. It controls the target applications using the Solaris debugger interface and detects the creation of child processes by overloading the fork() and exec() system calls. It also intercepts the signals SIGTRAP and SIGEMT, which are used by the sampling and watchpoint mechanisms described in detail below. The monitor process also contains data structures for storing the sampled information per target process and support functions for calculating miss ratios, reading debugger information etc. Several target processes running on the same computer may be monitored concurrently by a single monitor process.

3.4 The Sampling Algorithm

StatCache is based on reuse-distance sampling. One of the most important requirements of this statistical approach is that the samples are selected randomly. This means that every memory reference must be sampled with the same probability, and that the samples are selected independently of parameters like execution speed and system load. These requirements would be easy to meet if we had an exact access counter and could stop the execution exactly at any given memory operation. Unfortunately, the SPARC platform does not provide such a mechanism.

UltraSPARC III has 32-bit counters for counting executed instructions and load and store operations. The counters can be configured to generate the Solaris-signal SIGEMT on overflow, i.e. transition from all ones to zero. However, these counters have two limitations that prevent us from using them to select samples in a straight forward manner. Firstly, they cannot be configured to count both load and store operations simultaneously. Secondly, the overflow traps are deferred, which means that the processor may execute several instructions, sometimes twenty or more, after the actual overflow occurred, an effect also known as skid. The processor also tend to stop at certain instructions with much higher probability than other instructions that are executed the same number of times. An implementation that does not consider these hardware weaknesses would give very inaccurate results.

At first we tried to select samples using only the load counter. This actually worked quite well for applications with regular memory access patterns. However, when we ran complex codes, we found that applications with a varying load-store ratio caused our tool to select very unrepresentative samples. This forced us to instead use the method we describe below based on an instruction counter. The deferred traps were also difficult to handle. When we investigated the trap handling of the UltraSPARC processor, we found cases where the processor always stopped at the same instruction in a loop that consisted of over twenty instructions. The skid-compensation method introduced below solves this problem.

Figure 6 illustrates the final sampling mechanism. First, decide how far ahead we want to stop and take the next sample using a random-number generator.

$$NextSample := RandomGenerator() \quad (3)$$

Next, set the hardware instruction counter, $InstrCounter$, to generate a trap after $NextSample$ instructions, minus a small constant, $SkidComp$, which is larger than the maximum skid, i.e.,

$$InstrCounter := ICMAX - (NextSample - SkidComp),$$

where $ICMAX$ is the maximum value of $InstrCounter$. When the processor takes the overflow trap, StatCache reads

the current value of $InstrCounter$ to check how many instructions the processor executed after the overflow occurred and determines the number of instructions, $InstrRem$, that remain to execute until it reaches the actual sampling point. Thus,

$$InstrRem := SkidComp - InstrCounter. \quad (4)$$

The tool sets the target application in single-step mode and executes the remaining $InstrRem$ instructions to reach the intended sampling point. However, this instruction may not be a memory operation. There are two ways to handle this: the tool can either continue to single step until the next memory operation is found, or it can check whether the current instruction is a memory operation and take a sample only if it is. However, the first alternative does not fulfill our requirements of random sampling, because a memory operation preceeded by a large number of non-memory operations would be selected with much higher probability than a memory operation that is preceeded by just a few non-memory instructions. Thus, StatCache uses the second alternative. It decodes the instruction and if it is a memory operation, StatCache calculates the effective address of the instruction and determines the base address of the cache-line-sized piece of memory the instruction accesses, and passes the base address to the watchpoint mechanism for monitoring. If the intended sampling point is not a memory operation, the tool just skips it and restarts the sampling mechanism.

3.5 Hierarchical Sampling Scheme

The total number of sampling windows introduced in Section 2.2 could potentially become very large for long-running applications. However, the overall miss ratio is just the average of the miss ratios of the individual sampling windows, and the accuracy of an estimated average value depends mostly on the absolute number of samples and not the sampling ratio. Thus, there is no need to estimate the miss ratio of every such possible sampling window for long-running target applications, the accuracy would not improve much anyway. Furthermore, StatCache has the nice property that it does not consider cold misses, which enables the length of the sampling windows to be shorter than the warm-up period of a traditional cache simulator. This makes it possible to capture the cache behavior of the various phases of the target application using a large number of short sampling windows scattered throughout the target-application run. About twenty sampling windows are enough for applications with regular memory access patterns while complex codes require more sampling windows. We take advantage of this in our tool to reduce overall sampling rate, and thus reduce run-time, by introducing a hierarchical sampling scheme and use only a limited number of sampling windows scattered over the execution. This gives a fair estimate of the overall miss ratio with low run-time overhead. There are statistical methods that can determine the confidence interval of the results given the sample data and confidence level [4].

The hierarchical sampling scheme is easy to integrate in the sampling algorithm described in the previous section. We configure the random generator used to assign new values to the $NextSample$ variable (Equation 3) to produce smaller numbers if inside a sampling window, and a larger number at the end of each sampling window to produce an inter-window sampling gap.

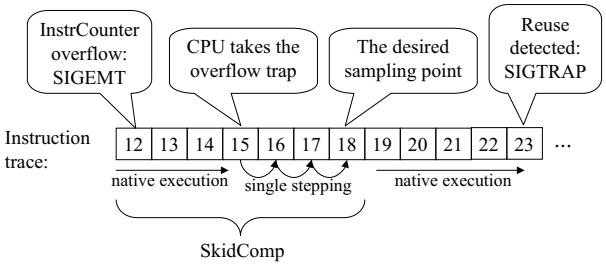

Figure 6: An illustration of the sampling and watchpoint mechanisms. The InstrCounter overflows at instruction 12, but the processor does not take the trap (SIGEMT) until instruction 15 because the trap is deferred. The sampling mechanism advances the target application instruction by instruction until the desired sampling point, instruction 18, is reached. If this is a memory operation, a watchpoint is set, and the next access to the same cache line is detected.

3.6 Watchpoint Mechanism

The cache-line-sized pieces of memory selected by the sampling algorithm must be monitored until the processor accesses them again. StatCache uses the watchpoint system provided by the operating system for this task. When a sample is selected, the data collector sets a watchpoint on the corresponding cache-line-sized piece of memory. This causes the operating system to send a SIGTRAP signal to the target process when it attempts to access the monitored memory. The signal is intercepted by the data collector, which records the reuse and removes the watchpoint. Figure 6 illustrates how the sampling and watchpoint mechanisms work together to select samples and detect data reuse.

The watchpoint mechanism in Solaris is based on page-wise memory protection using the MMU. The run-time overhead would therefore be very high if the target application often accessed memory on the same memory page as a monitored piece of memory. Thus, it is important to keep the number of active watchpoints low. Our statistical method can produce accurate results at a very low sampling rate, which keeps the average number of active watchpoints to a minimum.

3.7 Measuring Reuse Distance

The sampling and watchpoint mechanisms described above enable us to measure the reuse distance. The basic idea is to read the value of a memory reference counter when a sample is taken and check how much it has increased when the reuse is detected. However, measuring reuse distance also turned out to be tricky on SPARC, because it cannot count both load and store operations simultaneously.

Our profiling tool uses the load counter and a store-per-load scale factor to work around the limitation. The reuse distance, D, can be decomposed into load and store operations respectively, i.e., $D = L + S$. If we only know the number of load operations, L, but not store operations, S, we can instead approximate the reuse distance using L and an average number of stores per load, $StoreLoadRatio$. Thus, $D = (1 + StoreLoadRatio) \cdot L$.

The *StoreLoadRatio* is estimated per sampling window by counting store and load operations executed when the target application is in single step mode. This is a rough estimate if the *StoreLoadRatio* also varies a lot within each sampling window, but our results show that it is good enough for our tool. However, a hardware counter that counts both load and store instructions could improve accuracy. It is important to generate a new *StoreLoadRatio* for each sampling window because it varies greatly between sampling windows for some applications, for example $SPECmcf$.

3.8 Accurate Code Annotation

It is desirable to point out which instructions and which lines in the source code that cause the most cache misses. This is often troublesome using hardware counters because advanced hardware is needed to record exactly which instruction that caused a cache miss. The statistical approach enables an easy way to accurately annotate source code with cache misses.

Consider one sampling window. The sampling mechanism halts the processor at some random instruction, the *sample instruction*. It then checks if it is a load or store instruction, and if it is, calculates the effective memory address. The watchpoint mechanism monitors the cache-line-sized piece of memory accessed by the *sample instruction* until another instruction, the *trigger instruction*, accesses the same piece of memory again and records the reuse distance, D. When the tool has estimated the miss ratio of this sampling window, R_{window}, it can also estimate the number of cache misses between sample instruction and trigger instruction as $D \cdot R_{window}$. Formula 2 gives the probability that a datum still resides in the cache after n cache misses. Thus,

$$p = f(D \cdot R_{window})$$

gives the probability that the trigger instruction causes a cache miss.

Our tool records the PC (program counter) of both the sample instruction and the trigger instruction. It then aggregates the information per source code line using debugger information from the executable file and the object modules and presents it as miss ratio numbers next to the source code in the user interface as Figure 5 shows.

3.9 Identifying Data Structures

Identifying the data structures that cause poor cache utilization is often just as useful as identifying lines of code that cause many cache misses. A method similar to the one used for code annotation can be used also for data annotation. Besides recording reuse distance, sample and trigger instruction PCs, the tool also records the effective data address. Using a mapping between data structures and address ranges, the tool can easily calculate the miss ratio of any frequently accessed data structure.

There are different methods to find a mapping between data structures and address ranges. Statically allocated data may be identified using debugger information. However, many applications use mostly dynamically allocated data structures. One way to keep track of dynamically allocated data is to overload memory allocation functions, for example malloc(), and aggregate information about all data structures allocated at the same line in the program. A third way is to identify the name of the data structure by instruction PC (Program Counter). However, StatCache is currently based on the stabs debugger format, and stabs does not provide enough detail to implement the last method.

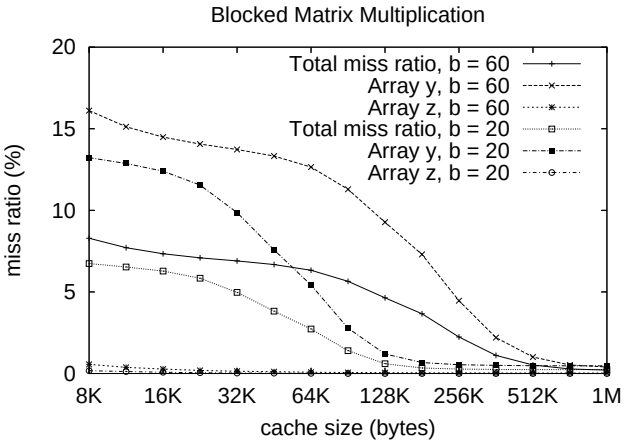

Figure 7: The tool allows individual arrays to be analyzed. The graphs show miss ratios for the arrays y and z as well as total miss ratio.

```
for (i = 0; i < ARCHnodes; i++)
  for (j = 0; j < 3; j++)
    disp[disptplus][i][j] +=
        2.0 * M[i][j] * disp[dispt][i][j] -
        (M[i][j] - Exc.dt / 2.0 * C[i][j]) *
        disp[disptminus][i][j] -
        Exc.dt * Exc.dt *
        (M23[i][j] * phi2(time) / 2.0 +
        C23[i][j] * phi1(time) / 2.0 +
        V23[i][j] * phi0(time) / 2.0);

for (i = 0; i < ARCHnodes; i++)
  for (j = 0; j < 3; j++)
    disp[disptplus][i][j] =
        disp[disptplus][i][j] /
        (M[i][j] + Exc.dt / 2.0 * C[i][j]);
```

Figure 8: Nested loops from SPEC CPU2000 *equake*

We illustrate the data structure identification in Figure 7 with a blocked matrix multiplication, $x = y \cdot z$. Statistics for x are not shown because it is rarely accessed. (The result of each vector times vector multiplication is stored in a register). The array y has the largest miss ratio, because it is referenced in column major order. The figure shows graphs for two block sizes, 20 and 60. Note how the smaller block size has moved the knees of the graphs to the left compared to the larger block size. For more advanced codes, this should turn out useful to track data structures with unfavorable access patterns.

3.10 Cache Data Reuse

Figure 8 shows two loop nests from SPEC *equake*. A software developer may want to know if any of the data accessed in the first loop nest survives in the cache until it is accessed in the second loop nest. This question could be answered using StatCache. All information we need is already calculated as described in Section 3.8. First, find all samples with a *trigger instruction* that belongs to the second loop nest.

Then put all these samples into buckets depending on what source line the *sample instruction* belongs to and sort the buckets in descending order. The first bucket indicates from which source line most of the data is reused, and so on. This information is readily available in our tool. A mouse click on a source line immediately displays this information in the user interface. Figure 5 shows the data reuse numbers for SPEC *swim* line 265. It shows that most data reuse, 88%, is self reuse, i.e. previously touched by the same line. The rest of the data accessed by line 265 is previously touched on lines 402 and 317. The tool could also estimate the probability that the data still resides in the cache. Section 3.8 describes how to calculate the miss probability of each trigger instruction. The average miss probability of the trigger instructions in a bucket tells if the data reused from the corresponding source code line still resides in the cache or have to be fetched from memory again.

4. EVALUATION

A tool such as the one described in this paper should be evaluated both for its speed and accuracy of its model.

4.1 Model accuracy

We evaluate the accuracy of the tool by comparing it to a trace-driven functional cache simulator. We use an in-house code instrumentation tool to generate traces that are fed directly to a cache simulator, hereafter called the *reference simulator*. The reference simulator simulates a fully associative cache with random replacement and line size 32 bytes. We have performed all experiments on a Sun V880 with two 750MHz UltraSPARC-III CPUs and 4GB memory. All benchmarks are compiled using gcc version 3.4.2 with optimization level 3, except *gcc* that did not compile properly with optimization. All benchmarks are simulated to the end.

The benchmarks come from the SPEC CPU2000 suite and they are *ammp*, *art*, *gcc* with input 166.i, *gzip* with inputs source, random and graphic, *mcf*, *twolf*, *vpr route* and *vpr place*. While StatCache includes the cache effects of library code, our reference simulator does not. Thus, these benchmarks are chosen because they do not spend much of their execution time in library code to enable a fair comparison. Many of these benchmarks have a cache behavior that varies greatly over time and some are very complex integer codes, like *gcc*.

Figure 4 shows the miss ratio as a function of cache size for ten selected benchmarks. The cache size is varied between 8KB and 4MB, and the figure shows two graphs for each benchmark, one generated by StatCache, and one generated by the reference simulator. The figures shows that StatCache captures the shape of the graphs well which we believe is an important property of a data-locality analyzer. This helps a programmer answer questions like: Does my data set fit in the cache or should I try blocking? or: I reordered the loops in this function, did it improve cache utilization? The tool will also tell if a code change had any positive effect on data locality even if it did not affect cache behavior on the actual development machine.

The benchmark *gzip* is a good example of how our profiling tool may be used to analyze the impact of different input data on cache behavior. A comparison of the graphs for *gzip source*, *gzip random* and *gzip graph* shows that the miss ratio for small cache sizes is much higher for the source input than for the other inputs. The miss ratio of the benchmark *gzip* is difficult to estimate because the benchmark compresses the input data several times at increasing compression levels. This causes the miss ratio to increase over time. We realized this when we experimented with the profiler and saw the miss ratio increase during the execution, but StatCache still manages to produce accurate results.

The differences we do see between the reference simulator and our tool have several causes. The first problem is that the reference simulator does not handle library code except `memcpy` and `memset`. Even though we have selected benchmarks that spend a small fraction of their execution time in other library code, many benchmarks still spend a few percent of their execution time in functions that are not traced by our reference simulator. This causes some of the deviations.

The next source of error is the sampling rate. Too few samples may cause large random errors. The tool reports the number of samples and a user can therefore handle this type of error by increasing the sampling rate. We know from experience that about 25 sampling windows is a minimum for complex codes like *gcc*, while it often suffice with less sampling windows for iterative numeric programs. The overall sampling rate in this evaluation is about 1 to 5,000,000, but the exact sampling rate varies between the different benchmarks because they have different instruction mixes. We describe in a technical report how the statistical method bootstrapping can be used to estimate the sampling error and find confidence intervals [3].

A third source of error is the sampling algorithm. Based on experience gained during the development of the tool we believe that this is the largest source of error. While it at first sight may seem easy to stop the processor at random instructions and sample, this really turned out to be one of the major problems. The method described in section 3.4 is the best we are aware of today, but we believe improvements are possible. The inherent error from the statistical method StatCache itself should be of minor importance because the theoretical results reported using a simulated implementation of StatCache show even smaller errors, also for integer benchmarks. [4].

4.2 Performance

A profiling tool should be fast to allow large workloads to be analyzed. Table 1 shows the run times for the benchmarks in our study with and without our profiling tool. Most of these applications have a rather short run time, and for such run times an overhead of up to 100 percent may be acceptable. Our profiler is well below that limit. For long-running applications we expect much lower relative overhead if the sampling rate is decreased proportionally. Furthermore, while StatCache is interactive, a user may view intermediate results at any time during an experiment and may choose to interrupt the application as soon as she has the desired information. This could shorten the turn-around time even further. The overhead for producing intermediate results is negligible.

There are different reasons for this run-time overhead. One of them is the watchpoint mechanism. While the page size of the SPARC architecture (8KB) is much larger than the cache line size, the OS kernel must handle a lot of MMU traps caused by memory accesses to pages that contain a watchpoint. Thus, the watchpoint mechanism is currently

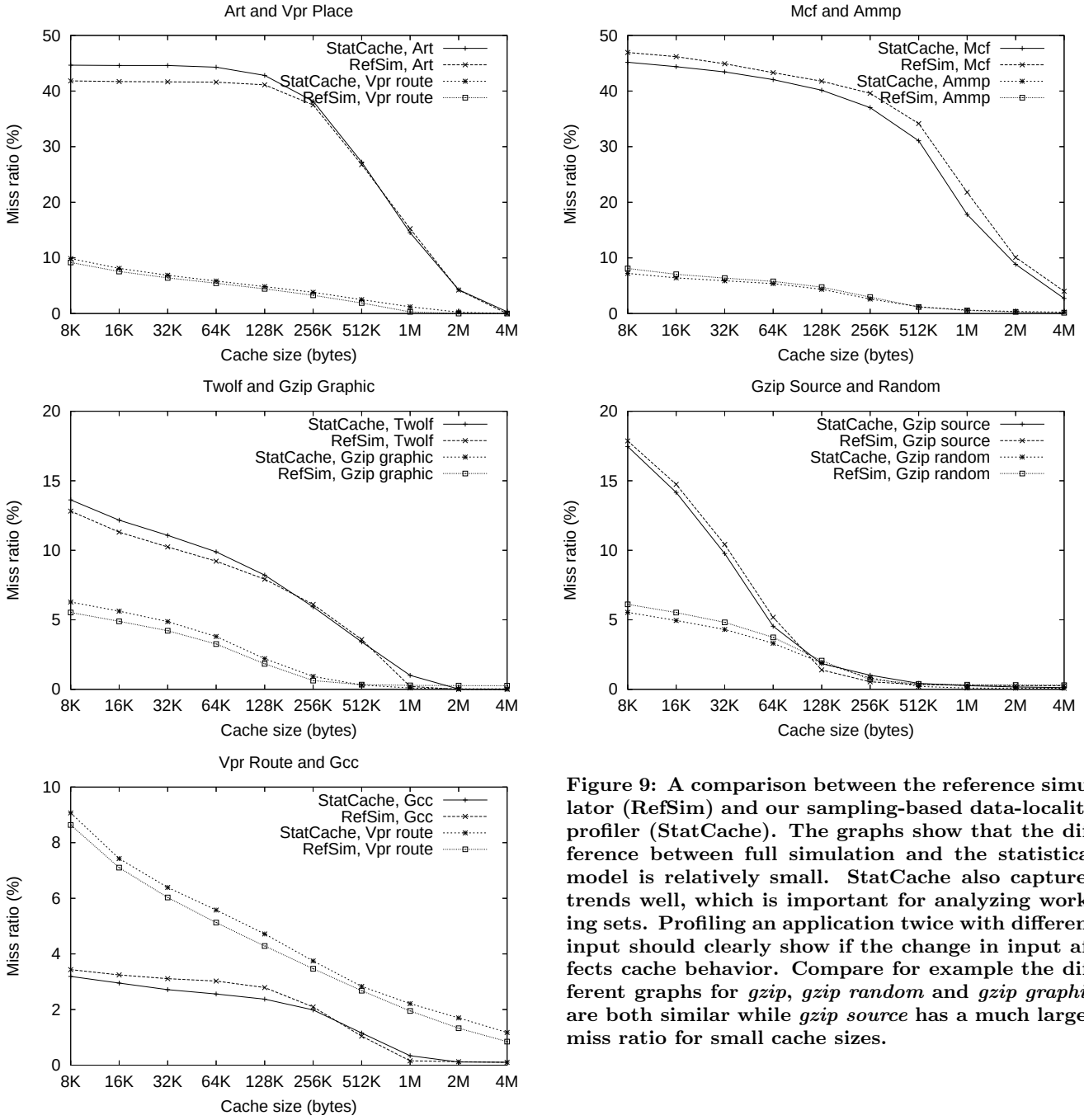

Figure 9: A comparison between the reference simulator (RefSim) and our sampling-based data-locality profiler (StatCache). The graphs show that the difference between full simulation and the statistical model is relatively small. StatCache also captures trends well, which is important for analyzing working sets. Profiling an application twice with different input should clearly show if the change in input affects cache behavior. Compare for example the different graphs for *gzip*, *gzip random* and *gzip graphic* are both similar while *gzip source* has a much larger miss ratio for small cache sizes.

the largest source of run-time overhead. Using the two hardware watchpoints available on the UltraSPARC III to handle watchpoints on frequently accessed memory pages would have reduced the run-time overhead considerably. This, however, requires kernel code modifications. The second most important overhead source is the sampling mechanism. StatCache must switch to single-step mode and single step about 50 instructions for each sample to compensate for trap skid and get representative samples. This consumes a lot of CPU cycles and causes a lot of context switches. We believe that an improved sampling mechanism could improve performance as well as accuracy.

5. DATA LOCALITY OPTIMIZATION

A fast flexible cache analysis tool can provide valuable insight in program behavior. This section discusses possible use of a fast tool with the capabilities described above and give a few examples of what kind of information that can be retrieved. [1]

[1]The results in this section are based on a pre-study code-instrumentation-based version of the tool that uses the same sampling algorithm.

Benchmark	Memory references (billions)	Run time (min/sec)	Run time with tool (min/sec)	Overhead (min / sec)	Number of sample windows	Overhead (%)
gcc	48	4m12s	6m55s	2m43s	37	64%
gzip source	16	4m31s	5m30s	59s	22	21%
gzip random	21	5m14s	2m36s	2m38s	22	50%
gzip graphic	24	5m10s	6m46s	1m36s	26	32%
art	14	4m44s	5m58s	1m14s	16	26%
ammp	131	21m4s	29m17s	8m13s	95	39%
vpr place	36	5m41s	8m47s	3m6s	33	54%
vpr route	31	5m20s	8m13s	2m53s	21	48%
mcf	19	11m0s	14m13s	3m13s	25	29%
twolf	100	19m28s	24m19s	4m51s	95	25%

Table 1: Run-time overhead of our profiling tool. The overhead includes everything, including graph generation. We have measured wall-clock time on a lightly loaded system.

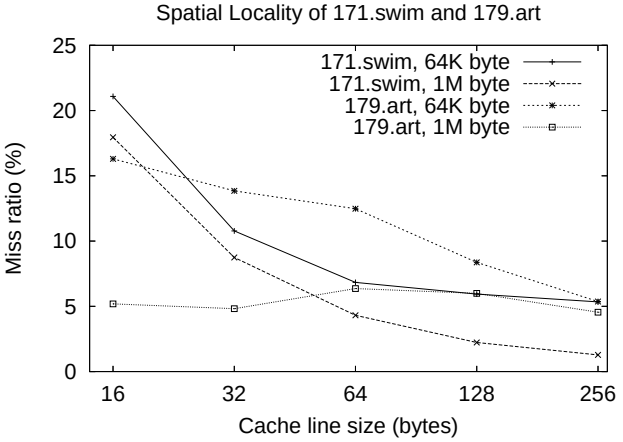

Figure 10: Spatial locality of 179.art and 171.swim. The graphs show the miss ratio of the two benchmarks as a function of the line size for the cache sizes 64K byte and 1M byte. Art has some spatial locality for the 64K byte cache but none for the 1M byte cache. Swim shows a very high degree of spatial locality for both cache sizes.

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

5.1 Spatial Locality in 171.swim

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 11 shows one of them. The arrays are larger than most caches, which means that very few array elements will survive in the cache until the next iteration. Thus, we expect almost only spatial locality, especially in small first level caches. 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 10 shows the miss ratio estimates of 179.art and 171.swim as function of the cache size. Swim behaves as expected. The cache miss ratio approximately halves when the cache line size doubles. The exception is large cache lines and a 64KB cache, were the miss ratios are very similar for the cache 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 some 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.

5.2 Quantifying Spatial Locality

Comparing the miss ratio of caches of the same size, but with different cache line sizes, allows quantification of spatial locality. A miss ratio that drops by 50 percent when the cache line size is doubled indicates good spatial locality. Let B_{small} be the size of the smallest data unit we consider, and let B be the cache line size. Let R_{small} be the miss ratio of a cache with cache line size B_{small}, and R be the miss ratio

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	188.ammp	-0.02
173.applu	0.97	301.apsi	1.01

Table 2: Spatial Use for eight 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.

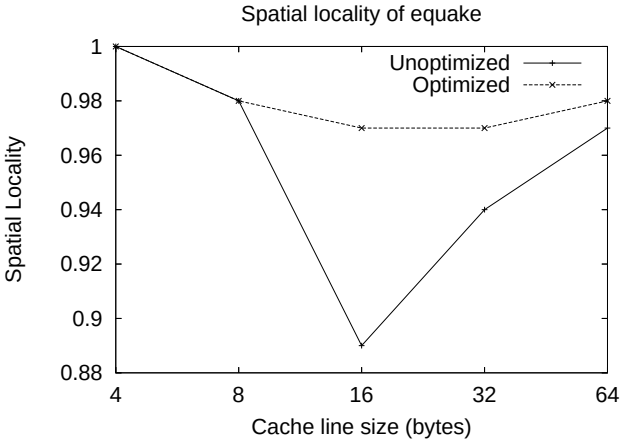

Figure 12: Spatial locality of SPEC 183.equake. The graphs show the Spatial use defined in Section 5.2 as a function of B (cache line size) for the original (unoptimized) and optimized code. This experiment use $B_{small} = 4$ and $B = 4$ through 64. The spatial locality of the optimized version is close to one for all line sizes, while the spatial locality for the unoptimized version drops for 16 byte lines.

of a cache with cache line size B. For a given cache size, define the spatial use:

$$Spatial\ Use(B) = \frac{1 - R/R_{small}}{1 - B_{small}/B}$$

The Spatial Use will be one if the cache miss ratio is inverse proportional to B, and zero if it is unaffected by B. Table 2 shows the spatial use for the eight SPEC benchmarks we investigate in this study. The values in the table are based on $B_{small} = 16$, and $B = 64$. 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.

5.3 SPEC 183.equake

It has been shown that equake suffers from poor spatial locality due to ineffectively allocated data [2]. In the original version of the application, the memory is allocated in small chunks (6 bytes large) with unused spaces between them. The paper showed that the application could run up to 40% faster with an optimized memory allocator. We were curious to see if our tool could guide a programmer towards this performance bug. We compared the original application with a version with an optimized memory allocator to see if our prototype tool and the spatial locality metrics would reveal the poor spatial locality. We varied the cache line size, B, between 4 and 64 bytes and calculated the spatial locality metrics defined above relative to the small 4 byte cache line. Figure 12 shows the results. 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. This study shows that we can find the same spatial locality problems with our prototype tool as the profiling tool described in [2] based on functional cache simulation.

6. RELATED WORK

There is a variety of methods to perform cache behavior studies. These include simulation, hardware monitoring, statistical methods and compile-time analysis. Compile-time analysis tools [35][8] estimate cache miss ratios by statically analyzing the code and determine when cache misses occur. Compile-time analysis major advantage is that it doesn't require the program to be executed, and can potentially be parameterized in terms of workloads etc. Its drawback is that it is limited to relatively well-structured codes where for example loop limits are known at compile time.

Cache simulators may be driven by instrumented code[13, 14, 20, 21, 23, 26, 27], on source code [17] or machine code levels, or the cache simulator incorporated in a full system simulator[24][22]. Their major limitation is their large slowdown. Simulation-based analysis can possibly combined with sampling (see below) to reduce the runtime overhead.

Cache-sampling techniques include set sampling and time sampling. In time sampling a cache model simulates continuous sub-traces from the complete memory reference trace. This is explored in papers [11, 15, 18, 19, 36]. It works well for smaller caches, but the need for long warmup periods makes time sampling less suitable for large caches. The problem of selecting statistically representative samples is explored in Perelman et al.[32] Set sampling is another approach, were only a fraction of the sets in a set-associative cache is simulated [11, 18]. It generally suffers from poor accuracy and can only be used as a rough estimate.

More recently, sampling guided by phase detection has been proposed [31, 37]. The idea is based on the observation that most applications have different phases during their execution. Within each phase, the system performs in a fairly invariant (often repetitive) way. Guided by phase detection algorithms, very sparse samples can still provide a representative behavior for the entire execution. While most work on phase-guided sampling has been targeting detailed pipeline simulation, similar techniques could also be applied to memory system modeling. Cutting down the number of samples for time-sampling of caches could turn out to be especially valuable, since the need to warm the large caches requires so many memory operations per sample. Phase-detection could also work well together with our tool. Phase detection could guide us to sample more or less often during the execution which could cut back on out runtime overhead further. The fact that we do not need to warm the caches before our model is valid further speaks in our favor.

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 [6] which is a common programming interface to access hardware monitoring aids. Histogramming and tracing hardware may be used to detect for example cache conflicts [30] and locate problem areas [7]. 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. Trap-driven trace generation has also been suggested [34]. 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[9] [10] and help choose between optimizations such as tiling, prefetching and padding[29]. Abstract cross-platform models for analyzing and visualizing cache behavior exist [25, 5, 38], mostly based on a reuse distance definition similar to the stack distance [28].

7. CONCLUSIONS

We have presented a fast and flexible tool for analyzing application cache behavior. The paper shows that it is possible to implement a fast profiling tool based on reuse distances sampling using only user-level operating-system support. The profiling tool is interactive and runs native unmodified application. We belive that the tool fills the gap between detailed but slow simulators that often force the analyst to reduce runs, and hardware monitoring with its limited flexibility. The speed of the tool enables execution with full data sets, while adding less than 40 percent overhead to the native execution time for long-running applications. This reduces many errors originating in unrepresentative and reduced experimental setups, at the expense of a less details.

The paper shows working-set graphs, spatial locality measures and presentation of data-centric information as examples of information that may be monitored with the tool. We also show how to produce detailed and accurate data-locality information at the source code level and how to find data reuse between different parts of the code. There are many details that cannot be explained in a paper and we plan to release the tool under an open source license to enable anyone interested to check any detail of the tool.

Acknowledgment

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

8. 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] 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.

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

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

[6] 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.

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

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

[9] 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.

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

[11] 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.

[12] Intel Corporation. Intel VTune Performance Analyzers `http://www.intel.com/software/products/vtune/`.

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

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

[15] 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.

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

[17] 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.

[18] 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.

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

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

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

[22] 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.

[23] J. Maebe, M. Ronsse, and K. De Bosschere. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Compendium of Workshops and Tutorials. Held in conjunction with International Conference on Parallel Architectures and Compilation Techniques.*, September 2002.

[24] 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.

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

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

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

[28] 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.

[29] 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.

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

[31] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for accurate and efficient simulation. In *Proceedings of SIGMETRICS*, 2003.

[32] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *In Proceedings of Parallel Architectures and Compilation Techniques*, 2003.

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

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

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

[36] D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. *ACM SIGMETRICS Performance Evaluation Review*, 19(1), May 21-24, 1991.

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

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