

Using Platform-Specific Performance Counters for Dynamic Compilation

Florian Schneider and Thomas R. Gross

Laboratory for Software Technology
Department of Computer Science
ETH Zürich
Zürich, Switzerland

Abstract. Hardware performance counters provide information about events in the hardware platform (e.g., cache misses, pipeline stalls), in contrast to profiles that capture program properties (e.g., execution frequencies for basic blocks, methods, function calls). As platform architectures become more complex and also more diverse, it is important for a compiler to exploit platform-specific information. A dynamic (JIT) compiler is in the unique position to run on the same platform as the target application, but in practice, exploiting the wealth of information available through performance counters is far from easy. If a JIT compiler is to use performance counter information, this information must be fine-grained (e.g., attributing cache misses to a single load instruction) and must be obtainable without undue overhead. We present a runtime+compiler framework to tie hardware performance counter information to a dynamic compiler and argue that the overhead is low and fine-grained. As parallel architectures or multi-core architectures proliferate, performance issues will play a crucial role in all compilation engines, and our paper reports on a modular approach to make such counter information available to the compiler.

1 Introduction

The combination of VM and JIT compiler is now the most common execution platform for programs written in object-oriented languages. Unlike the classic ahead-of-time compilation model, the JIT compiler is able to take immediate advantage of dynamic information. There are two kinds of information that such a compiler may use: *profiles*, i.e. measurements of program properties (e.g., number of method invocations) and measurements of platform-specific properties, such as number of cache misses, TLB misses, branch prediction failures). The latter must be obtained from the performance measurement unit of the execution platform, and this paper details how this information can be provided to and used by a compiler for a high-level language like Java.

Many compilers for high-performance linear algebra computing already use information from the execution platform for cache optimization: For example, blocking is a common technique to reduce cache misses in matrix computations,

but using it effectively requires that the characteristics of the memory hierarchy be considered [14]. Another example is inter-variable padding [18], which can be used to reduce conflict misses, but requires a precise knowledge about the application's memory access patterns. For programs in the domain of scientific computing these access patterns can often be obtained exactly or can be approximated by an analytical model. Still, the design of performance monitoring units is still the subject of current research [15].

On the other hand, object-oriented programs have many properties that are difficult to determine at compile-time (e.g., memory access patterns, synchronization patterns). As multi-core and parallel architectures proliferate, attention to performance for object-oriented programs increases the need to use platform-specific information to generate efficient code. Most modern CPUs (like the Pentium 4 (P4), Itanium, PowerPC) offer the ability to deliver information about performance-related events to the OS or the application, yet most previous JIT compilers focussed only on using program properties to guide optimizations [6]. Preliminary studies (without full compiler support) have however demonstrated that platform-specific metrics can also improve the performance of object-oriented programs[2].

To be useful for an optimizing JIT compiler the collected information must be accurate enough and cheap to obtain at run-time. Since modelling memory access patterns analytically for pointer-intensive code (typically found in OO programs) is not feasible at the moment, the use of hardware performance monitors presents a viable way of getting detailed information about memory hierarchy performance aspects. This paper presents a general infrastructure to feed hardware performance monitor information into a JIT compiler at run-time.

2 Requirements

Our basic assumption is that the object-oriented program executes on some VM and that this VM provides a JIT compiler (possibly offering different optimization levels). A module that makes information from the hardware performance monitors available in a JIT compiler must meet a couple of requirements:

- The runtime+compiler infrastructure should be flexible enough to allow obtaining different execution metrics. The exact group of events that can be monitored depends on the specific hardware performance counters that are available, but the interface between compiler and performance monitoring unit should attempt to hide machine-specific details where possible.
- The overhead to obtain the monitor's information should be low, and the executed applications should not be perturbed by the measurements.
- Processing the information should be done in a separate module, to keep the need for changes to VM and/or the compiler to a minimum.
- The information must be accurate enough to be useful for online optimizations in a JIT compiler. Often the granularity of a method or even a basic block is too coarse to allow the compiler to infer what instruction/operation is responsible for some event (e.g., cache misses).

- The platform should work for “general” VMs. We don’t want to change the core VM code too much. Otherwise the effort to port it to another VM would be prohibitively large.

Of course, any compiler that uses platform-specific information may also use profile information, e.g., to decide where and when to exploit the results obtained from the performance measurement unit. We will not dwell on this aspect in this paper.

3 Related work

There are two areas of prior work that we concentrate on in this paper: techniques to provide platform-specific information in a form that the compiler can exploit and specific optimizations in a compiler that are influenced by this information. While there exists a fair bit of prior work regarding profiling (e.g., discussion of types of profiles, algorithms to select the best place to insert code to maintain counters, choice of sampling intervals), it is not central to the topic of this paper, and so is not covered here.

3.1 Data gathering techniques

Profiling to obtain execution frequencies and profile-guided optimizations have been applied in ahead-of-time compilers (see, e.g., [17, 8]) and JIT compilers [6, 20]. Here we focus on related work that uses hardware-specific information for optimizations.

Ammons et al. [5] use hardware performance counters together with path profiling. They use code instrumentation to associate hardware metrics (like cache misses) to basic blocks and execution paths in the program. The reported overhead of flow and context sensitive profiling is between 60 and 80%. This overhead is acceptable when doing off-line performance analysis.

Trace-driven simulation of the memory hierarchy can be used for analyzing data locality and identifying bottlenecks [11, 10]. The results depend on how precise the simulation reflects the real platform. One disadvantage of precise simulation is that the slowdown can be several orders of magnitude [23].

Vera et al. [24] use an analytical model to approximate the behavior of the CPU and memory hierarchy. They use cache miss equations to describe the behavior of loop-oriented code. Their approach is mainly targetted at scientific computations which exhibit regular access patterns.

In recent years OO applications have been analyzed using profiles and hardware support. Hauswirth et al. [12] analyze Java programs and their interaction with the VM, the OS and the hardware using *vertical profiling*. They distinguish different execution layers in a system: application, libraries, VM, OS and hardware. To analyze the performance of these layers they introduce “software performance monitors”. These monitors capture performance characteristics of the different subsystems. The results are correlated with data from the hardware performance counters to find out how different metrics influence each other.

Georges et al. [9] present an off-line technique for analyzing the performance behavior of individual methods. Since instrumenting every method would be too expensive, they identify method-level phases by measuring the execution time spent in each method. In a second step, only those methods that constitute an execution phase are instrumented. The hardware performance counters are read at the method prologue and at the epilogue. Finally, the profiling results are mapped back to the Java source code. The approach has a low overhead because only those methods selected by the phase analysis are instrumented. It uses the hardware performance counters in normal counting mode, not in event-based sampling mode like we do.

3.2 Optimizations

Cache optimizations reduce the gap between memory and processor speeds. Loop-tiling, loop-skewing, and blocking [25] can increase data locality in scientific, array-oriented programs. To obtain maximal performance, cache parameters must be considered when choosing the block size [14].

Software-controlled prefetching [16] hides the memory latency by overlapping memory access with other operations. It is mainly used for scientific applications which are array-oriented and have regular iteration patterns that can be determined statically.

OO programs require a different approach because they usually use pointers heavily and do not exhibit the regular structure of scientific applications. Adl-Tabatabai et al. [2] use hardware performance monitors of the Itanium 2 processor to inject prefetch instructions into Java programs. Their approach relies on the fact that objects that are accessed consecutively often have a constant delta between their addresses. A “meta-data graph” captures references between classes that exhibit a large number of long-latency misses and the corresponding deltas. The prefetching uses this graph to ensure the right data is available in the cache. They achieve a speedup of 14% for the SPEC JBB2000 benchmark [21]. Software prefetching is very effective on Itanium because it has only in-order execution and lacks the hardware-based prefetching of the P4.

Huang et al. [13] implemented a technique called online object reordering that reorders objects at garbage collection time. They identify “hot” fields by gathering access statistics using code instrumentation. The garbage collector then copies the object referenced by hot fields together with their parent object to increase spatial locality.

4 Implementation platform

This section presents background of the hardware and software platform that we used for our implementation.

4.1 Hardware performance monitors

The P4 offers a large variety of performance events for counting [1]. Two modes of operation are supported:

- Normal counting: The performance counters are configured to count events detected by the CPU’s event detectors. A tool can read those counter values after program execution and report the total number of events. This mode can be used to obtain numbers like the cache miss rate, total execution cycles, and so on. One application would be to evaluate the effect of program transformations.
- Sampling-based counting: Whenever a certain number of events has occurred, the CPU samples its register contents. This way it is possible to locate the sources of an event. The P4 supports precise event-based sampling, so it reports both the exact instruction where the sampled event happened and the register contents at that point.

To keep the overhead of sampling low, the CPU stores a certain number of samples in a buffer provided by the OS. The CPU generates a performance monitor interrupt when this buffer is filled up to a “high-water” mark. The interrupt service routine of the OS copies the samples to a more permanent location.

This mechanism makes it possible to obtain data address profiles with the P4. The instruction pointer (IP) together with the other registers’ contents can be used to calculate the data address of an event (e.g., cache miss). A data memory address of an event can be computed by decoding the instruction that caused the event and using the values of the registers to calculate its address operand.

Previous CPUs could only measure an approximate location for sampled events because of a super-scalar design and out-of-order execution. The P4 and other newer architectures (e.g. Itanium) have the capability to localize the event precisely (precise event-based sampling). Sprunt [19] wrote a detailed overview of the P4’s hardware performance monitoring capabilities.

4.2 Jikes RVM

Our implementation is done with the IBM Jikes RVM (version 2.3.3) [4, 3], a high performance Java virtual machine written mostly in Java. It includes an adaptive optimization system [6]. First, every method is compiled with a simple and quick baseline compiler. Only methods that are executed frequently enough are recompiled and optimized further.

5 Runtime+compiler platform issues

Our extension allows the VM to monitor the performance of a running application using the CPU’s hardware performance monitors. In a dynamic compilation

environment like the Jikes VM the compiler can then react and use this information to dynamically recompile and optimize parts of the program. We extended the abyss&brink tools [7] to configure and access the P4 performance counters. The tools consist of a kernel module and a user-level program to gather statistics about the program that is being measured.

The kernel module initializes the hardware performance monitors and provides the sampling interrupt handler that copies the samples from the kernel buffer into a more permanent buffer supplied by the application (in our case the Java VM). The P4 hardware supports precise event-based sampling for only a subset of events. The most important of those are:

- L1 and L2 load misses,
- DTLB misses, and
- branch mispredictions.

At the moment the type of event that is monitored is specified as a command-line parameter.

We modified the Linux kernel and the kernel module to be able to monitor individual processes. Otherwise the results would be disturbed by other processes running at the same time.

The monitoring infrastructure consists of three parts:

1. Loadable kernel module: The kernel module offers the functions to access the performance counter hardware. It is implemented as a device driver, and the application communicates with it via IOCTL calls. The kernel module hides the platform-specific details from the JVM. It also provides the interrupt handler that is called by the sampling hardware when the CPU buffer for the samples is full. When this happens the samples are copied into a more permanent location. At the moment we allocate a 4MB shared memory buffer for this purpose.
2. Native shared library (C): Since we cannot call device drivers directly from Java or from the Jikes RVM we use a native library to provide an interface and call it via the Java Native Interface (JNI). The library gives access to the shared buffer where all the collected samples are stored.
3. Collector thread (Java): We use a separate Java thread that polls the device driver via the library interface whether there are any new samples. The polling interval is set to 1-10ms depending on the size of the sample buffer. Each sample is converted into a Java object by the collector thread and handed to the VM for further processing.

Figure 1 shows how the samples get from the CPU to the JVM. Buffering the samples in user space makes the JVM independent of any platform idiosyncrasies as those are handled by the kernel device driver.

On the P4 platform one sample has a size of 36 bytes. It contains the instruction pointer (IP) where the sampled event occurred and all the values of the registers at this point in the program. Figure 2 shows the structure of one sample. The CPU writes those values directly into the buffer provided by the

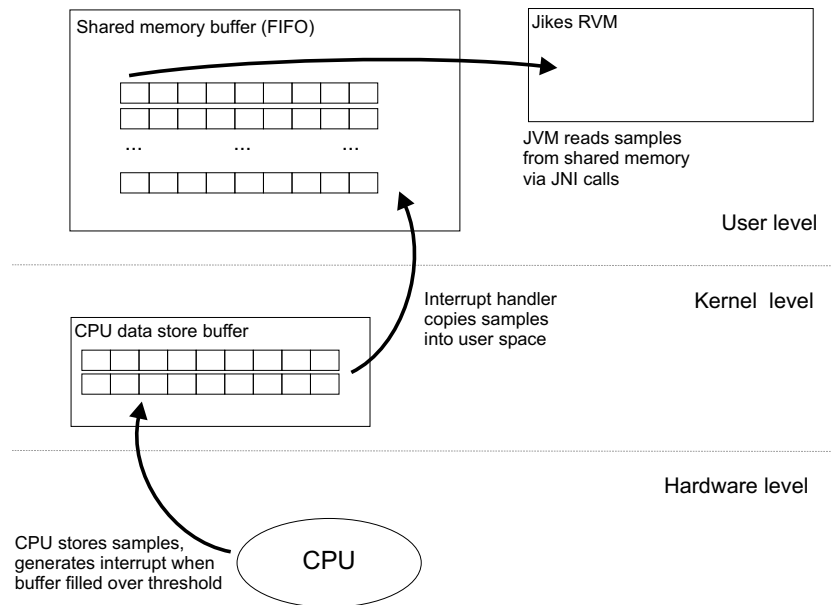


Fig. 1. Getting the samples from the CPU to the JVM.

kernel module. To be able to use these raw data for optimization we need to recover some higher-level information for each sample. The JIT compiler keeps a sorted table of all methods' start/end addresses that were compiled so far. From the IP we can quickly find out the method and the bytecode instruction where the event happened by performing a binary search. When the adaptive optimization system recompiles a method this method table must be updated.

The bytecode instruction tells us the operation and the type of the object that caused the event. With the bytecode instruction we can produce a human readable output that contains the Java statement and the source line number for the event. When we encounter an event caused by a heap access we determine the actual type of the responsible object by scanning backward in memory starting at the calculated data address until we find the object header [2].



Fig. 2. One sample (total 36 bytes) contains the instruction pointer (EIP) and all register contents.

6 Evaluation

6.1 Measurements

The measurements are carried out on a 3.0 GHz P4 processor running the Linux kernel version 2.4.26. This processor has a 1MB L2-cache and a 64K L1 cache and 1024 MB of main memory. One cache line has a width of 64 bytes. For each data point we ran the benchmark three times and reported the average. As a VM we ran Jikes RVM 2.3.3 with the configuration “FastAdaptiveGenCopy”, which includes the adaptive optimization system [6] and a generational garbage collector.

For our experiments we set the system to measure cache misses. The sampling interval should be large enough to keep the overhead low, but not too large. Otherwise the collected data won’t be meaningful. A sampling interval between 1000 and 10000 events proved to be most suitable for our benchmark programs.

6.2 Sampling overhead

Table 1 compares the performance of the system with and without sampling enabled. For this measurement we sampled every 10000 and every 1000 events. (columns $s=10000$ resp. $s=1000$). For the SPEC JVM98 [22] and the SPEC JBB2000 [21] benchmarks we observed an overhead between 0.1% and 2% (average 1.6%) for a sampling interval s of 10000. For $s = 1000$ the overhead is between 0.1% and 5% (average 2.1%).

program	orig	s=10000	s=1000
javac	7.18	1.02	1.02
raytrace	4.04	1.02	1.02
jess	2.93	1.01	1.00
jack	2.73	1.00	1.03
db	10.49	1.01	1.03
compress	6.5	1.01	1.02
mpegaudio	6.54	1.02	1.00
jbb	6209.67	1.02	1.05
average		1.016	1.021

Table 1. Overhead of collecting sample data with two different sampling intervals $s=1000$ and $s=10000$.

To analyze the influence of the sampling interval on the overall performance we studied the JBB benchmark in more detail. Figure 3 shows the correlation between the sampling interval and the performance as measured by the *specJBB* score. The interrupt rate of the HPM hardware grows linearly with decreasing sampling interval size. The overall performance drops even more at high interrupt rates; the resulting context switches for each invocation of the interrupt service

routine consume further CPU time. From the observed execution times we can estimate the cost of processing one sample with $< 1\mu\text{s}$ (=3000 CPU cycles on a 3 GHz P4).

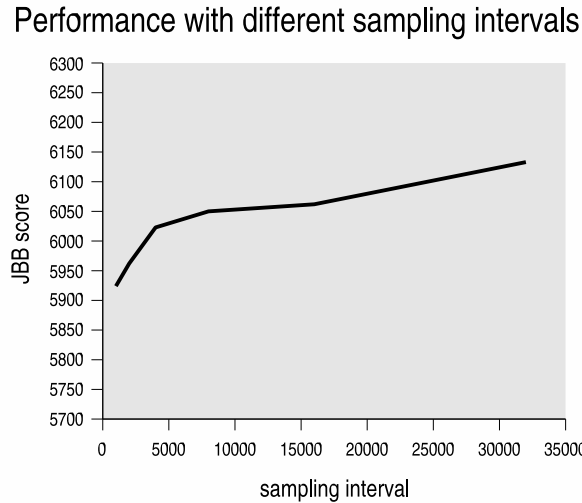


Fig. 3. Performance with different sampling intervals

6.3 Distribution of cache misses

The precise event-based sampling of the P4 allows us to measure the distribution of cache misses over the load instructions in the program. We use a sampling interval of 1000 events for L2 misses and 10000 events for L1 misses.

For *db*, *javac*, and *specJBB* we measure the frequency of L1- and L2-misses. Figure 4 shows the histogram of the 100 most contributing load instructions for L1 cache misses. These loads produce 37% of the L1 misses in *javac*, 98% in *db*, and 55% in *specJBB*. The picture is different for long latency L2 cache misses. Figure 5 shows the same information for L2 misses. There, the 100 most contributing load instructions are responsible for 74%, 99% and 85% of the events. For *db* the distribution of L1 and L2 misses is quite similar – there are very few “hot” loads. In *javac* and *specJBB*, on the other hand, the L1 misses are generally distributed over the whole program, whereas the L2 misses are more localized. (except for one instruction in *specJBB* that produces the majority of the L1 misses). This suggests that if we focus optimizations on these cache misses, or “hot” spots, we can achieve a large impact.

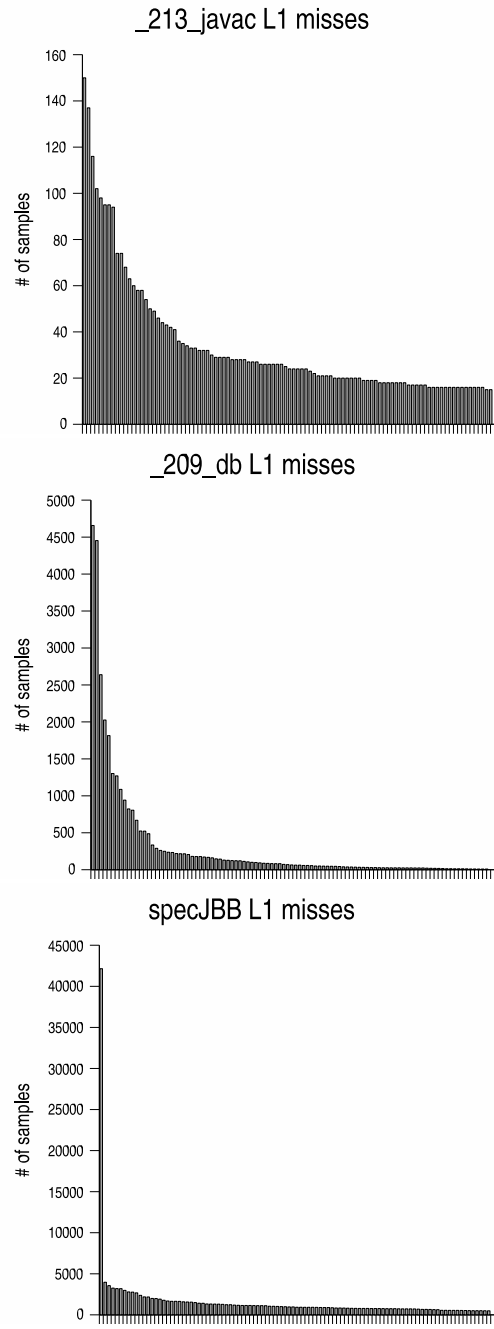


Fig. 4. Histograms of L1 cache misses (100 most contributing load instructions).

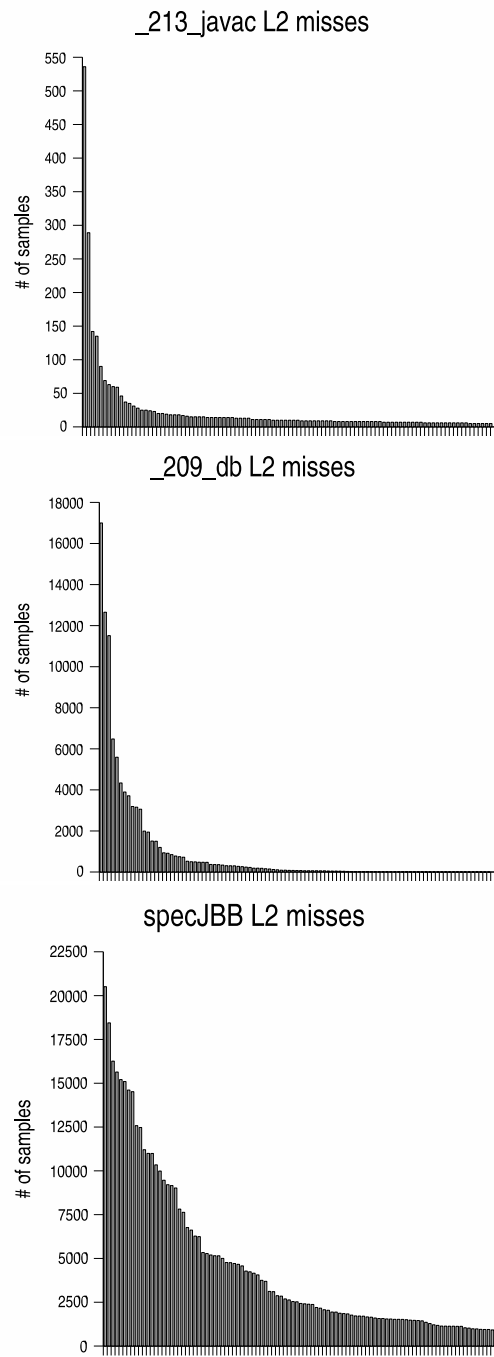


Fig. 5. Histograms of L2 cache misses (100 most contributing load instructions).

7 Concluding remarks

Using platform-specific information about a program's execution in a dynamic compiler is attractive; the result of the platform's performance measurement unit can be mapped to source-language constructs that are relevant for the compiler. A necessary condition (to be satisfied by the platform's architect) is that the monitoring unit can accurately capture the processor state related to an event. Fortunately, newer processors provide this capability.

A JIT compiler is in a good position to exploit this information, since we have shown that the overhead of gathering and processing the information about hardware-specific events can be kept low.

To demonstrate the practicality of this approach, we implemented a module to tie the Jikes VM to the execution monitoring unit of the P4. As an example application we showed that the compiler can use this mechanism to identify individual load instructions that are responsible for a high percentage of the cache misses. This information allows feedback-driven optimization that does not solely rely on high-level information like method execution frequencies, but is directly guided by information about performance critical hardware events.

8 Acknowledgments

We thank Lukas Löhner and Flavio Pellanda for their help and their contributions to the implementation.

References

1. IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide. 2005.
2. A.-R. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *Proc. of the ACM SIGPLAN 2004 Conf. on Programming language design and implementation*, pages 267–276, New York, NY, USA, 2004. ACM Press.
3. B. Alpern, C. R. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. L. ber, T. Ngo, M. F. Mergen, J. C. Shepherd, and S. Smith. Implementing jalapeno in java. In *Conference on Object-Oriented*, pages 314–324, 1999.
4. B. Alpern, D. Attanasio, J. Barton, M. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, T. on Ngo, M. Mergen, V. Sarkar, M. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. rini Srinivasan, and J. Whaley. The Jalapeno virtual machine. *IBM Systems Journal, Java Performance Issue*, 39(1), 2000.
5. G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proc. of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96, New York, NY, USA, 1997. ACM Press.
6. M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeo jvm. In *Proc. of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–65, New York, NY, USA, 2000. ACM Press.

7. Brink & Abyss. http://www.eg.bucknell.edu/bsprunt/e-mon/brink_abyss/brink_abyss.shtm.
8. P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21(12):1301–1321, Dec 1991.
9. A. Georges, D. Buytaert, L. Eeckhout, and K. D. Bosschere. Method-level phase behavior in java workloads. In *Proc. of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 270–287, New York, NY, USA, 2004. ACM Press.
10. A. J. Goldberg and J. L. Hennessy. Performance debugging shared memory multiprocessor programs with mtool. In *Supercomputing '91: Proc. of the 1991 ACM/IEEE conference on Supercomputing*, pages 481–490, New York, NY, USA, 1991. ACM Press.
11. S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In *Proc. of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 146–157, New York, NY, USA, 1993. ACM Press.
12. M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: understanding the behavior of object-oriented applications. In *Proc. of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 251–269, New York, NY, USA, 2004. ACM Press.
13. X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: improving program locality. In *Proc. of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 69–80, New York, NY, USA, 2004. ACM Press.
14. M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of block algorithms. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, CA, Apr. 1991.
15. Lubeck, O. et al. WS6: Hardware Performance Monitor Design and Functionality, Los Alamos Computer Science Institute Symposium 2005. Web archive <http://lacs.rice.edu/workshops/hpca11>, Feb 12-16 2005, San Francisco, 2005.
16. T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proc. of the 5th international conf. on Architectural support for programming languages and operating systems*, pages 62–73, New York, NY, USA, 1992. ACM Press.
17. K. Pettis and R. Hansen. Profile guided code positioning. In *Proc. ACM SIGPLAN'90 Conf. on Prog. Language Design and Implementation*, pages 16–27, White Plains, N.Y., June 1990. ACM.
18. G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proc. of the ACM SIGPLAN 1998 Conf. on Programming language design and implementation*, pages 38–49, New York, NY, USA, 1998. ACM Press.
19. B. Sprunt. Pentium 4 performance monitoring features. In *IEEE Micro*, pages 72–82, July–August 2002.
20. T. Sukanuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *Conf. on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '01)*, pages 180–194, 2001.
21. The Standard Performance Evaluation Corporation. SPEC JBB2000 Benchmark. <http://www.spec.org/jbb2000/>.

22. The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. <http://www.spec.org/osg/jvm98>, 1996.
23. R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: a survey. *ACM Comput. Surv.*, 29(2):128–170, 1997.
24. X. Vera, N. Bermudo, J. Llosa, and A. González. A fast and accurate framework to analyze and optimize cache memory behavior. *ACM Trans. Program. Lang. Syst.*, 26(2):263–300, 2004.
25. M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proc. of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 30–44, Toronto, Ontario, Canada, June 1991.