





© 2006 Sun Microsystems, Inc. All rights reserved. The SML Technical Report Series is published by Sun Microsystems Laboratories, of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

#### TRADEMARKS

Sun, Sun Microsystems, the Sun logo, Java, Java HotSpot, Java Native Interface, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@sun.com>. All technical reports are available online on our website, <http://research.sun.com/techrep/>.

# *Introspection of a Java Virtual Machine under Simulation*

Greg Wright

Sun Microsystems Laboratories  
Mailstop UMPK16–159  
16 Network Circle  
Menlo Park CA 94025  
*greg.wright@sun.com*

Phil McGachey\*

Dept. of Computer Sciences  
Purdue University  
250 N. University Street  
West Lafayette IN 47907-2066  
*phil@cs.purdue.edu*

Erika Gunadi\*

Dept. of Electrical & Computer Engineering  
University of Wisconsin–Madison  
1415 Engineering Drive  
Madison WI 53706  
*egunadi@ece.wisc.edu*

Mario Wolczko

Sun Microsystems Laboratories  
Mailstop UMPK16–158  
16 Network Circle  
Menlo Park CA 94025  
*mario.wolczko@sun.com*

\* Work performed during an internship at Sun Microsystems

## **1. Introduction**

Modern commercially-significant object-oriented programming systems, such as Sun Microsystems' Java™ and Microsoft's C#, are based upon virtual machines. Applications to be executed on the virtual machine (VM) are distributed in a platform-independent format (Java bytecodes [7] or Microsoft Intermediate Language, MSIL), simplifying the development and deployment of applications on heterogeneous networks.

The virtual machine, in this case, is a piece of software which intermediates between the applications and the underlying operating system and hardware, typically dynamically compiling application code to the host's instruction set and invoking platform-specific libraries. The separation provided by the VM thus allows greater scope for operating system and hardware architects, freeing them from the constraints of machine-code binary compatibility. In theory, a new machine can be designed together with a VM to improve the combined behavior of the system.

The VM, however, interferes with common performance analysis methods. For example, the mapping of program counter values back to instructions and source information is complicated when the code is compiled and perhaps recompiled or relocated during the execution.

We have constructed a tool which extracts high-level information from a Java HotSpot™ virtual machine running inside the Simics full-system simulator [8]. It obtains information such as: Is a given virtual address in the heap? If so, which object? Which

field? Of what class? For program counter values in generated code, we can derive symbolic information all the way back to approximate Java source line numbers. We obtain this information without modification of the VM sources and without disruption to the state of the simulation, a significant improvement over existing techniques.

The remainder of this paper is structured as follows. We first explain our motivation in more detail. Following that is a description of the tool itself, including our methods for extracting data and its limitations. We conclude with an experiment which demonstrates the kind of studies enabled by the tool.

## 2. Background & Motivation

One attractive feature of systems based on virtual machines is the freedom to evolve the hardware and software together without the need to preserve the architectural interface. The virtual machine is a point of high leverage for system engineers. Improvements in the VM's compiler technology, for example, may benefit all applications with a single upgrade. Similarly, memory management and garbage collection, historically re-implemented in each separately-compiled application, can be provided as a service to all applications. Hardware/software co-design is common in the embedded space, but it is not frequently practiced in the commercial server area. One reason is that it disrupts the established procedure for designing computer systems. Independent software vendors (ISVs) are very reluctant to distribute machine-code binaries optimized for every platform, because of the difficulty of maintaining and qualifying all the versions. Thus hardware is commonly designed as a drop-in replacement for an existing system, with the aim of running existing user-mode applications without recompilation. Hardware architects are accustomed to taking instruction-level traces of applications, characterizing their behavior in terms of memory reference patterns, branch predictability and so on, and designing a new machine optimized for these characteristics, with performance estimated using trace-driven simulation. With the interposition of a VM, however, many of these characteristics can be customized or indeed adapted at run time – they are no longer properties peculiar to the application but are moderated by the virtual machine.

Consider, for example, a virtual machine designed for a simple processor with a 'classic RISC' short single-issue pipeline and a small instruction cache. On such a machine the penalty for indirect branches may be low, with priority given to limiting the size of generated code. In this case the VM may compile method calls as indirect jumps through virtual method tables (v-tables). An instruction trace taken from this VM may lead the designers of a subsequent CPU, with a deeper, more complex pipeline, to optimize indirect branches, perhaps by adding branch target buffers and multiple levels of speculation. Yet the characteristics could be changed completely with the introduction of inline caches [1] and method inlining, thus converting some indirect branches to direct, eliminating others, and placing quite different demands on the hardware.

Ideally, then, the design of virtual machines and hardware platforms would proceed in concert, each informed by the choices inherent in the other. Current commercial systems must of course also balance the performance of legacy statically-compiled applications,

but we do not consider them further here on the grounds that their characterization is much better understood. Some experiments may be performed on existing systems, using performance counters or sampling, as long as the existing and proposed systems are sufficiently similar. Other experiments may be carried out under simulation; execution-driven simulation allows feedback between the proposed architecture’s state and virtual machine activity in a way which trace-driven simulation does not. Both of these methods produce very low-level data, whether of individual events, for example the program counter values of loads which caused cache misses, or in aggregate, such as the total number of cache misses.

Here we pursue execution-driven simulation, so that we can non-disruptively inspect the state of the simulated machine. That is, we want to be able to look inside and examine the contents of the caches, say, or the CPU’s branch prediction information, without executing code on the system and disturbing that state. Execution-driven simulation has had a reputation for being slow, but modern systems such as Simics may have a slowdown of only around 15–20x, at least in a warm-up mode with caches disabled (see later). While running on a real machine may offer some of the same information, if interfaces are available which expose the desired state, sampling can trade disruption off against completeness. At the cost of increased simulation time, our technique potentially offers complete information without any disruption, and for hardware arbitrarily different from existing machines.

To gain understanding of the system’s behavior, one must relate the low-level events back to the activity of the VM and application. Simics gives very good visibility into the hardware state at the level of instructions, registers and physical addresses, but symbols, classes and methods are more meaningful for the VM and application programmer (Fig. 1). For example, a given static load instruction may frequently miss in the cache, but is it part of the VM binary or is it in generated code compiled by the VM? If it is in generated code, is the target a field in an object, and if so, of what class and which field? Such questions are not easy to answer under simulation; interpreting the raw machine state requires considerable knowledge about the operating system (process and virtual memory mappings) and virtual machine data structures. The following section describes how we obtain this information.

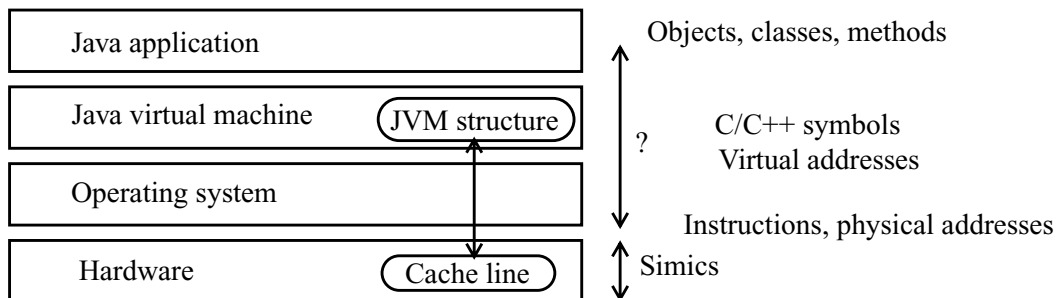


Fig. 1: Abstraction layers

### 3. The HotSpot Serviceability Agent in Simics

Our tool runs as a module inside the Simics execution-driven simulator, and examines a Java HotSpot virtual machine running on a simulated SPARC® system with the Solaris™ operating system. Solaris and the HotSpot executable are not modified in any way: we do not recompile the VM to indicate its state or report events. Instead, we rely on knowledge of the layout of the HotSpot data structures, including VM structures (i.e., C++ objects in the HotSpot source code) and Java objects.

#### 3.1 The HotSpot Serviceability Agent

Our tool is based on the HotSpot Serviceability Agent (SA) [11], which was developed by the HotSpot product team for its own internal use. In its original role, the SA is attached to a HotSpot process or core file via a system debugger interface, for example the Solaris dbx debugger or /proc pseudo-filesystem. The SA has knowledge of the layout of the C++ data structures in a particular version of HotSpot, and can navigate around the memory image by following pointers from known locations.

For our purposes we can consider the Serviceability Agent as being composed of three layers of code. At the bottom, some platform-specific code sits between the higher layers and the Solaris debugger interfaces (dbx or the /proc pseudo-filesystem). The functionality required here is small: reading words from virtual memory locations, and obtaining the virtual addresses of certain symbols in the HotSpot shared library (libjvm.so). These static locations, identified by C/C++ symbols, are the roots for the traversal of the HotSpot data structures.

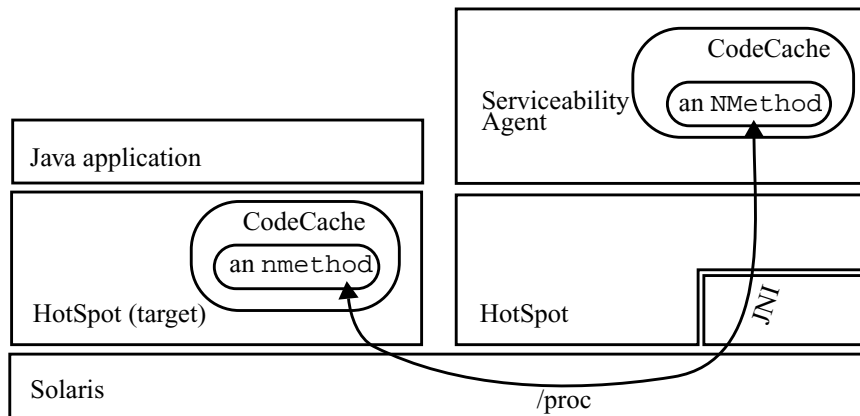


Fig. 2: The Serviceability Agent

The middle layer of SA code, the majority, is written in Java, and replicates the data structures in a particular version of HotSpot. For example, in the HotSpot source a piece of runtime-generated code is represented by a C++ object of class `nmethod` (Fig. 2). In the Serviceability Agent there is a corresponding Java class `NMethod`. An `NMethod` instance serves like a proxy for an `nmethod`, and has accessors for all of an `nmethod`'s state; the `NMethod` instance is filled out by reading bytes from the HotSpot process's virtual mem-

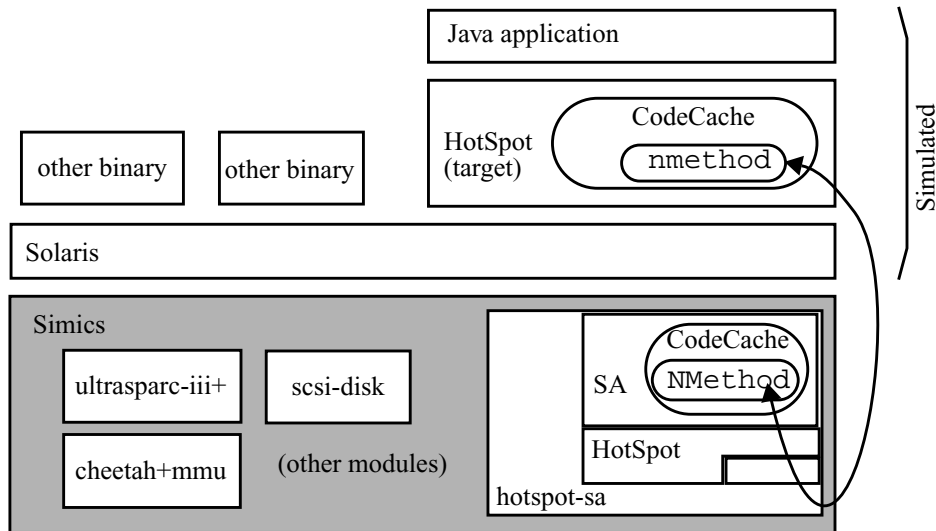


Fig. 3: The Serviceability Agent inside Simics

ory image, calling the platform-specific debugger through the Java™ Native Interface (JNI).

The top layer of the SA is a set of command-line and graphical tools through which the user can interactively display and investigate the state of the HotSpot process, as mimicked in the SA’s Java representation. We do not use this code, as we do not currently present a user interface when running under simulation; instead we query the HotSpot state programmatically, for example in the event of a cache miss.

### 3.2 Adapting the SA

We run the Serviceability Agent inside a ‘hotspot-sa’ Simics module, alongside the existing CPU, cache, disk, etc. modules (Fig. 3). The middle layer of the SA, the part which embodies information on the HotSpot data structures, is unchanged, and runs inside a JVM ‘embedded’ in the Simics module (i.e., our Simics module is linked with the HotSpot library, and creates a JVM using JNI routines). All of our modifications are at the lowest level, replacing the SA’s dbx or /proc back-end with code to extract state from inside the Simics simulator, using the standard Simics APIs and inter-module communication.

The two essential low-level capabilities on which the SA is built are (1) reading words from virtual memory addresses inside the target HotSpot process, and (2) obtaining the addresses of symbols.

Reading data from a process inside Simics is complicated by the simulated operating system. Just like the hardware it is simulating, Simics has no knowledge of the OS’s internal data structures. It has virtual-to-physical address translations only for pages which are currently in its translation buffer; when a page fault occurs, the OS will provide an appropriate translation. We cannot assume that an address required by the SA will be recently touched in the simulation; indeed, the SA sometimes requires data from a page in the tar-

get HotSpot's address space which has not yet been brought in from disk, because Solaris lazily pages binaries into physical memory after they are mapped. We track changes to the page tables by watching TLB map and demap events, using the SPARC context identifier to associate changes with a particular process. In practice, as is common in benchmark configurations, the simulated OS is lightly loaded, so the target HotSpot process may be uniquely identified by its 13-bit context ID. It remains possible that in a heavily loaded system (i.e., thousands of processes) Solaris could reallocate the context ID, in which case our tool would produce incorrect results. However, we have never observed this behavior.

When the target HotSpot process is invoked we pass an environment variable to the Solaris dynamic linker (`ld.so`) requesting it to load an 'audit library' [13]. The audit library is notified by the dynamic linker as it populates the target process's address space: a callback function is invoked when a new shared object (`.so`) is loaded, and at that point we execute the Simics 'magic instruction', a no-op which causes an event to be reported to our Simics module. Our module receives the notification and records the shared object's path and the virtual address at which it was mapped into the HotSpot process's address space. As a side effect, we also discover the SPARC context ID of the HotSpot process from the CPU which executed the magic instruction.

The few extra instructions executed whenever a shared object is loaded are our only disruption to the execution of the target HotSpot virtual machine. In particular, there is no disruption once the Java application of interest has started up.

Given the shared object's path (on the simulated machine), plus the contents of the simulated disk image, we can recover the on-disk contents of the newly-loaded shared object. This gives us the virtual addresses of symbols in the library, used by the SA for navigating the HotSpot image, and also for mapping program counter values back to function names in C or C++ source. We also cache a copy of the on-disk contents of library segments mapped into the target HotSpot process's address space, to provide the contents should the SA request data from a page which has not yet been paged in. We do not currently track the eviction of modified pages from virtual memory to swap space, but paging of the target JVM does not occur in the benchmark studies of interest.

### 3.3 Limitations

Our tool has very good visibility into the data structures of the virtual machine. However, we are lacking equivalent abilities for the Solaris kernel and C libraries. We can reconstruct some required information, in particular virtual memory mappings, and knowledge of the ELF shared object format gives us some basic symbolic information.

The most significant deficit is the ability to inspect thread and stack state. The debugger interfaces usually used by the SA ensure that the target HotSpot's threads are suspended and the thread and stack state are restored to the canonical representation specified by the API and calling conventions. Under simulation we can observe other machine states which are not observable in the real machine. For example, a thread's stack may be distributed between memory and register windows [16], and even a descheduled thread may still have state in register windows somewhere in the system. It is non-trivial to look at a CPU and

find out which thread-library thread or Java Thread is executing on it, or to recover the state of a given thread.

Such information could in theory be recovered with a hypothetical ‘Solaris kernel serviceability agent’ and ‘thread-library serviceability agent’. In their absence, we restrict ourselves to looking at the top stack frame of a running thread, i.e. the register contents of a particular CPU.

We also cannot obtain precise information when mapping program counter values in dynamically-compiled code back to Java source line information. The HotSpot compiler does not maintain such mappings for every generated instruction, and in general it is not a straightforward relationship because of the optimizations employed.

#### **4. Previous work**

Characterization studies of Java workloads have generally fallen into two groups. One approach has been to remain close to the Java level, investigating such properties as the relative frequencies of bytecodes [9], object lifetimes and demographics [2], or properties of Java method invocations [15]. This work may involve bytecode-level tracing of application behavior [17]. These studies have the advantage that the data obtained are independent of the underlying virtual machine, and thus remain valid for as long as the benchmarks used are representative of the workloads of interest. They have the disadvantage that not all metrics will imply characteristics of interest to the system designer. For example, the frequencies of individual bytecodes are arguably of marginal utility in the presence of a highly optimizing dynamic compiler.

Many other studies have examined the low-level behavior of various Java benchmarks on a particular virtual machine and platform, whether on real hardware or under full-system simulation [6], and data have been obtained on, for example, instruction profiles [5] and cache and TLB miss rates [4]. However, such work may date rapidly with improvements in VMs; compiler and garbage collection implementations are now very different from the Java 1.1 VMs used in many early studies. Our tool is also dependent on a particular VM version, although we aim to relate the observed effects back to higher levels, to obtain results of wider applicability. Porting our tool to a new version of HotSpot is straightforward, as our modifications are limited to the lowest level of the SA with no changes to HotSpot itself.

Although these studies measure the virtual machine’s effects, they have limited visibility into the VM. More information may be obtained by instrumenting the VM to record state transitions, for example when switching from a mutation to garbage collection mode [10]. This is complicated when transitions may appear in generated code, for example instructions in an inlined write barrier, or when the system does not transition cleanly from one mode to another, as with a concurrent garbage collector. The VM may also be modified to generate low-level traces during execution, by instrumenting generated code [12]. This can provide good information for the VM’s data memory references, for example field information even after optimizing compilation, but omits the behavior of the OS and disturbs the instruction stream.

Vertical profiling shares the goal of relating virtual machine behavior and architectural effects [3]. The VM is instrumented to include monitors, for example a (software-implemented) count of the number of fast-path lock acquisitions in generated code. Periodically the VM's counter values are sampled, along with CPUs' hardware performance counters (of, for example, instructions executed or data cache misses) and other software performance monitors provided by the operating system. The result is a data set spanning multiple levels of the system at each sampling point, and visualization tools help in correlating the behaviors observed at the different levels. Correlation does not imply causality, and the user can infer cause and effect only with a good understanding of the system. Vertical profiling has the advantage that it runs directly on real hardware, but like all such techniques it is disruptive to the execution and cannot observe hardware state for which no software interface is provided (for example, the current contents of the caches). The data serve to give a summary of the system's behavior over relatively long intervals (millions of instructions, for a millisecond sampling interval) without details on any individual event; in contrast, our simulation-based approach lets us obtain a great deal of information about individual operations, but at a high cost in execution speed.

Recent versions of the Sun ONE Studio Compiler Collection include performance tools [14] which can profile the execution of Java code running on the HotSpot virtual machine, extending the existing tools for C, C++ and Fortran code. The higher-level information is obtained with the cooperation of the VM, through profiling interfaces, and augments the data obtained on the VM binary itself. The caller/callee relationships are thus correctly reported for a call stack containing interleaved C/C++ and Java frames across the user application, library and virtual machine code; the Solaris 10 Dynamic Tracing (DTrace) feature adds kernel profiling too. As with any tool based on periodic sampling of the call stack there will be a small amount of disruption, although because the information is derived by sampling this will be significantly less than with vertical profiling's counters. Again, there is no direct observability of hardware effects other than the coarse CPU counters for instructions executed, cycles, etc., but the tools are intended for application and system programmers, not microarchitecture studies.

## **5. An experiment: cache contents and misses**

In all of the subsequent experiments our simulated system consists of an UltraSPARC III+ processor core with 512MB of main memory, a four-way associative 64kB L1 data cache (D-cache) with 32-byte lines, and a direct-mapped 1MB L2 cache ("external cache", or E-cache) with 64-byte lines. The CPU runs at 750MHz, which equals 750 MIPS in the absence of a timing model; we are simulating cache behavior, but not including miss penalties. The instruction stream is excluded from the cache traffic. Unless otherwise stated, the Java virtual machine is a production build of HotSpot 1.5.0 with no command-line flags used, running on the Solaris 9 operating system.

For our first experiment, we sample the contents of the caches every 10M instructions. Each cache line in the cache is classified as part of the Java heap, code generated by the dynamic compiler, other non-heap data structures (e.g. C++ objects), or belonging to

another process. The Java heap is subdivided into the new, old and permanent generations; the permanent generation contains objects such as `java.lang.Class` instances, which are not managed by the garbage collector.

We present results for three of the SPECjvm\_98 benchmarks, which have the advantages that they are widely used in the existing literature and also sufficiently trivial to be simulated to completion in a few hours. The three benchmarks, `javac`, `jack` and `jess`, are each reported as three graphs: the D-cache and E-cache contents, plus the heap occupancy for each generation. `Javac` is an early version of the Java source to class file compiler, `jack` is a parser generator, and `jess` is an expert system that applies inference rules to a database of facts.

The three benchmarks show very different behaviors on the graphs of cache contents. `Javac`'s cache contents, for the first three quarters of its execution, are heavily dominated by the VM data structures, with intrusions of heap objects around the garbage collection points (Fig. 4). A new-space (semi-space scavenging) collection appears as a sharp decrease in new-space usage atop a stair-step rise in old-space caused by the promoted objects; the full heap mark-sweep collections are drops in old-space size. After around 17B cycles the allocation rate (slope of the heap usage sawteeth) increases and the heap contents dominate the cache. There is sufficient application code that the cache behavior is dominated by the virtual machine's compiler, even when the default "client" compiler is used; although not graphed here, the heavily-optimizing "server" compiler, invoked by a command-line flag, extends run time significantly for a single run of these small benchmarks in a single-processor configuration. As seen in Fig. 5, `jack` exhibits more uniform behavior; there is a gradual increase in allocation rate as compilation is completed, but the GC survival ratio is very low and the heap barely dents the E-cache, except during full heap GCs, suggestive of an application working set of young objects much smaller than the VM's working set. The E-cache occupancy drop at 2.6B cycles shows where the VM was pre-empted for a daemon. `Jess` (Fig. 6), demonstrates the other extreme. Compilation is substantially finished by 1.5B cycles, and thereafter the behavior is a rapid turnover in new space with slow accumulation into old space. The cache occupancy graphs show that access to the longer-lived old-space data structures is significant; old objects generally occupy more of the D-cache than new objects.

For an alternative view of the cache behavior, we look at the program counter value of every instruction causing an E-cache miss. For instructions in the virtual machine we report C/C++ function symbols, except in the VM's interpreter loop where we instead obtain information on the Java method being interpreted. For runtime-generated code we report Java method information.

Fig. 7 shows the source of `javac`'s E-cache misses, for a short period (1.4B cycles) starting just before the first full-heap garbage collection. A large block of misses around 11.2B instructions caused by C and C++ code indicates the activity of the collector itself. During this time the mutator is mostly suspended, as shown by the lack of misses caused by interpreted and compiled code. Contrast this to Fig. 8, from a portion with little GC activity.

Here the application code causes the majority of misses, with only a small spike of C++ code at 15.1B instructions for a new-space collection.

Fig. 9 shows a section of Fig. 7, with more detail on the VM’s C++ methods. Each C++ method is classified by its place in the VM source tree as GC, Memory (heap management), Runtime, Compiler, Code (primarily managing generated methods in the code cache), or Other. Interpreted or compiled Java methods are shown as Application in this graph. The full-heap collection starts at 11.17B cycles with a large spike in cache misses (‘Code’); this is due to a traversal of the code cache to locate root references. The GC runs until 11.29B cycles, after which the application and compiler resume. It is interesting to note that, except for the spike in cache misses caused by the code cache traversal, the GC itself has a miss rate comparable with the rest of the execution. This effect was observed in all three benchmarks.

Fig. 10 shows misses to the E-cache while running a portion of the jess benchmark. This period occurs after the initial compilation phase has completed, so the vast majority of misses are caused by compiled code. A series of spikes in C and C++ code indicate new-space garbage collections. An isolated region of misses caused by other contexts is seen at 2.5B instructions. This is due to a context switch by the OS, and can be seen to align with the drop in cache occupancy for the VM process seen in Fig. 6. A full-heap GC follows shortly afterwards at about 2.6B cycles, then an isolated compilation phase at 2.8–2.9B cycles. In Fig. 6 the GC is evident as a marked increase in the permanent generation in the cache; the compilation then displaces all of the heap for VM data structures, and is also visible as a plateau in the allocation graph.

A breakdown of the top 10 classes for 1.5B cycles of javac, towards the end of the execution, is shown in Table 1. This table shows that the contributions made by the standard library classes to cache miss rates are significant, which suggests that these classes (particularly String and Hashtable) may be good targets for cache optimizations in the VM or library code. This is an opportunity for further investigation.

## 6. Conclusions

We have introduced a tool for obtaining high-level information from a Java virtual machine running inside an execution-driven simulator, and demonstrated some applications of it on three Java benchmarks. We have shown the contents of the caches, and also source-level information on which code is causing cache misses. We know of no previous studies which collected non-disruptive, exact data on cache behavior and mapped them back to the JVM and application level. Some observations result from these data: Full-heap garbage collections are disruptive to the cache state (as shown clearly in Fig. 5 and Fig. 6) but do not cause a large increase in average miss rate (compare Fig. 7 with Fig. 8). The disruptive effect of compilation, even late in the execution, is significant (Fig. 5 and Fig. 6). We therefore caution against using single runs of these small benchmarks for detailed performance analysis work. We also observed that standard Java library classes (Table 1) are responsible for a significant fraction of the cache misses, at least within the accuracy limitations of our tool (§3.3).

We believe that the kind of information produced by our tool will prove of value to system architects and virtual machine implementors, and encourage further studies relating observed effects back to VM and application behavior and interactions. Much further work remains, both in development of the tool and its application to larger, more complex, benchmarks.

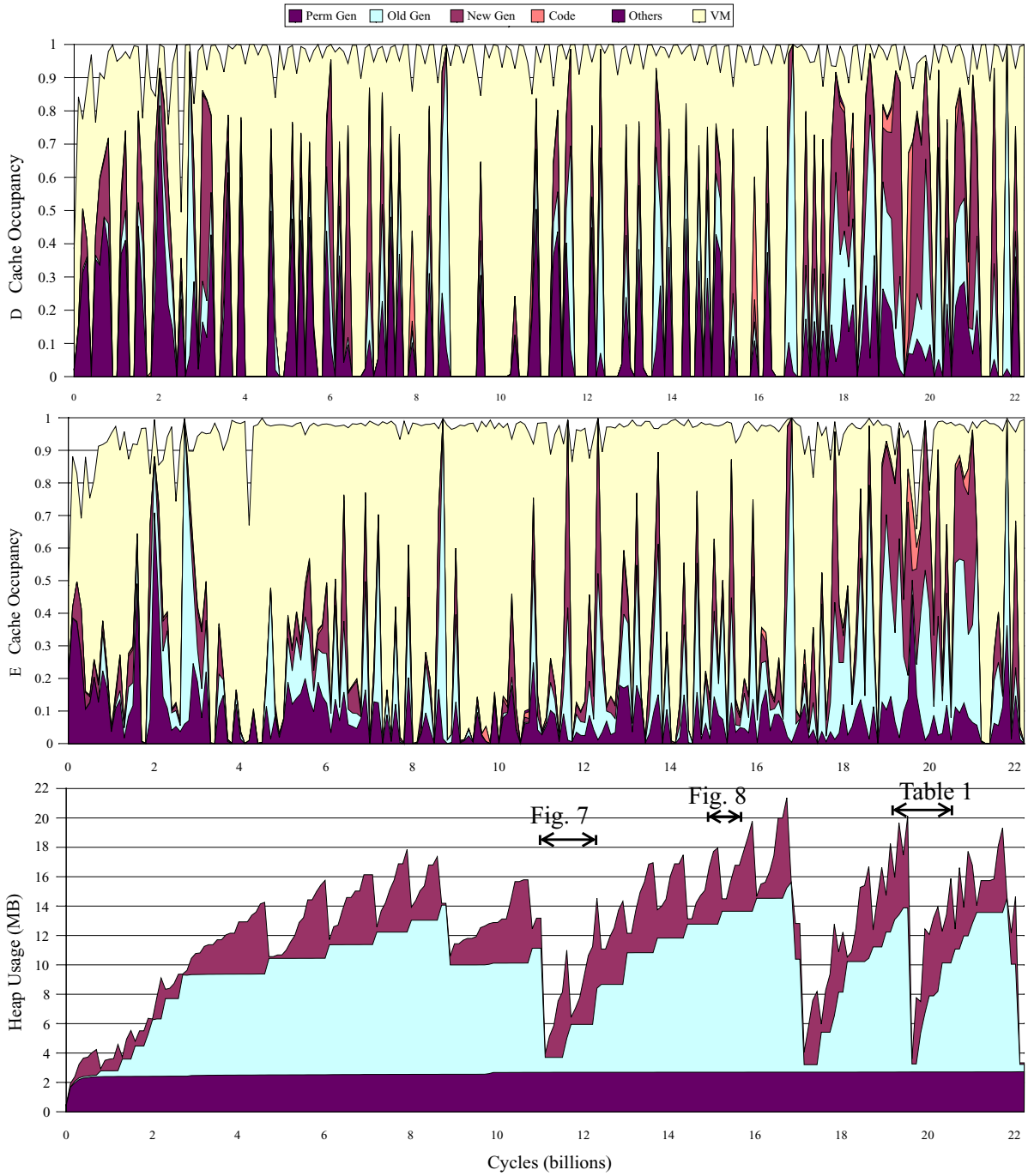
## Acknowledgements

We should like to thank Ken Russell for his help with the Serviceability Agent, Mike Paleczny for careful explanations of many aspects of the HotSpot virtual machine and comments on this work, and also Dan Nussbaum for reviewing this report.

## References

- [1] L. Peter Deutsch & Allan M. Schiffman. *Efficient Implementation of the Smalltalk-80 System*. Proc. 11th Symp. Principles of Programming Languages (POPL), pp. 297–302. ACM, Salt Lake City, UT, 1984.
- [2] S. Dieckmann & U. Hölzle. *A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks*. Proc. European Conf. Object-Oriented Programming (ECOOP), July 1999.
- [3] Matthias Hauswirth et al. *Vertical Profiling: Understanding the Behavior of Object-Oriented Applications*. Proc. 19th Conf. Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pp. 251–269. ACM, Vancouver, British Columbia, Canada. October 2004.
- [4] Martin Karlsson et al. *Exploring Processor Design Options for Java-Based Middleware*. Proc. Intl. Conf. Parallel Processing. Oslo, Norway. June 2005.
- [5] Jin-Soo Kim & Yarsun Hsu. *Memory System Behavior of Java Programs: Methodology and Analysis*. Proc. SIGMETRICS, pp. 264–274. ACM, Santa Clara, CA. June 2000.
- [6] Tao Li et al. *Using Complete System Simulation to Characterize SPECjvm98 Benchmarks*. Proc. Intl. Conf. Supercomputing (ICS), pp. 22–33. ACM, Santa Fe, NM. 2000.
- [7] Tim Lindholm & Frank Yellin. *The Java Virtual Machine Specification*, Second Edition. Addison–Wesley, 1999.
- [8] P. S. Magnusson et al. *Simics: A full system simulation platform*. IEEE Computer, 35(2), pp. 50–58, February 2002.
- [9] Ramesh Radhakrishnan et al. *Java Runtime Systems: Characterization and Architectural Implications*. IEEE Transactions on Computers, 50(2), pp. 131–146. February 2001.

- [10]Anand S. Rajan et al. *Cache Performance in Java Virtual Machines: A Study of Constituent Phases*. Proc. 5th IEEE Intl. Workshop on Workload Characterization, pp. 81–90. Austin, TX. October 2002.
- [11]Kenneth Russell & Lars Bak. *The HotSpot Serviceability Agent: An Out-of-Process High-Level Debugger for a Java Virtual Machine*. Proc. Java Virtual Machine Research and Technology Symposium (JVM’01). Usenix, Monterey, CA. 2001.
- [12]Yefim Shuf et al. *Characterizing the Memory Behavior of Java Workloads: A Structured View and Opportunities for Optimizations*. Proc. SIGMETRICS, pp. 194–205. ACM, Cambridge, MA. 2001.
- [13]Sun Microsystems, Inc. *Linker and Libraries Guide*, in the Solaris 9 9/04 Software Developer’s Collection. Part No. 817–3677–10. Santa Clara, CA. 2004.
- [14]Sun Microsystems, Inc. *Sun Studio 10: Performance Analyzer Reference Manual*. Part No. 819–0493–11. Santa Clara, CA. 2005.
- [15]John Waldron & Owen Harrison. *Analysis of Virtual Machine Stack Frame Usage by Java Methods*. Proc. IASTED Intl. Conf. Internet and Multimedia Systems and Applications (IMSA). Nassau, Bahamas, October 1999.
- [16]David L. Weaver & Tom Germond, eds. *The SPARC Architecture Manual*, Version 9. Prentice–Hall, 1994.
- [17]M. Wolczko. *Using a Tracing Java Virtual Machine to gather data on the behavior of Java programs*, <http://research.sun.com/people/mario/tracing-jvm/>, 1999.



**Fig. 4: javac D- & E-cache contents, and heap usage**











