VIProf: Vertically Integrated Full-System Performance Profiler

Hussam Mousa, Chandra Krintz, Lamia Youseff, and Rich Wolski
Dept. of Computer Science
University of California, Santa Barbara
{husmousa, ckrintz, lyouseff, rich}@cs.ucsb.edu

Abstract

In this paper, we present VIProf, a full-system, performance sampling system capable of extracting runtime behavior across an entire software stack. Our long-term goal is to employ VIProf profiles to guide online optimization of programs and their execution environments according to the dynamically changing execution behavior and resource availability. VIProf thus, must be transparent while producing accurate and useful performance profiles.

We overview the design and implementation of VIProf and empirically evaluate the system using a popular software stack – one that includes a Linux operating system, a Java Virtual Machine, and a set of applications. This composition is commonly employed and important for high-end systems such as application and web servers as well as Computational Grid services. We show that VIProf introduces little overhead and is able to capture accurate (function-level) full-system performance data that previously required multiple profiles and extensive, manual, and offline post-processing of profile data.

1. Introduction[1]

Recent advances in virtualization techniques expose a number of new opportunities for applications that execute using them. Virtualization systems are increasingly popular software systems that multiplex lower-level resources among higher level software programs and systems. Examples of virtualization systems include a vast body of work in the area of operating systems [22, 14, 7], high-level language virtual machines such as those for Java and .Net, and, most recently, virtual machine monitors (VMM) [28, 20, 1].

In our research, we investigate opportunities for performance optimization across the entire system stack (OS, Java Virtual Machine, application server, application), when we use virtualization to isolate system instances. Key to our approach is our assumption that a single application executes at a time. This assumption holds for both batched, cluster systems that execute scientific applications, as well as for a wide range of web, Grid service, and application server systems. Virtualization enables us to customize the system stack for a particular application, i.e., the application that is currently executing (or will execute once scheduled), very aggressively. Then, when the application completes, virtualization facilitates the replacement of the entire system stack with another one (e.g., for another application).

Given this execution model, i.e., isolated application instances that execute within guest operating systems over a virtualizing software layer, we can consider novel techniques for optimization and specialization across all layers of the software system. In particular, we are interested in automatic and dynamic adaptation, customization, and integration of the application, runtime, and operating system according to the dynamically changing characteristics of program behavior and underlying resource availability. Our approach is called VIVA – Vertically Integrated VirtualizAtion, and our preliminary results indicate significant potential and opportunity [12, 32, 31, 30].

As a first step toward enabling dynamic customization, in this paper, we investigate an efficient and accurate profiling system that captures performance data transparently *while* the application executes. Such information is vital for guiding effective runtime (re-)optimization and (re-) adaptation according to dynamically changing conditions.

Current profiling systems are limited in that they operate on a single layer of the execution stack at a time. Analyzing profiles and identifying common bottlenecks *across* layers (OS, JVM, application) is thus, currently not possible without manual, offline, and tedious post-processing of profiles collected via different tools. Moreover, even *across* layers, existing profiling techniques have difficulty capturing performance data on all code that executes, e.g. code that is dynamically compiled. If we are to optimize across the different layers of the software stack, we must be able to sample all code that executes, and interrelate the perfor-

[1]This work was funded in part by NSF grants CNS-0546737 and ST-HEC-0444412 and Microsoft.

mance data across all software layers. To address this need for a transparent, online, full-system profiling system, we present VIProf, a *V*ertically *I*ntegrated *P*rofiler.

VIProf is an extension of the system wide profiler, OProfile, for Linux [19]. We extend OProfile by integrating its sampling system to access program-level information from any Application Virtual Machine. We implement a VIProf prototype using the open source Jikes Research Virtual Machine (JikesRVM) from IBM T.J. Watson Research center. We evaluate the overhead and efficacy of this VIProf prototype and detail how VIProf captures full-system performance behavior.

In the sections that follow, we present the background and motivation for our research, as well as other related work in this area. We then describe VIProf in Section 3. In Section 4, we demonstrate the potential benefits of vertically integrated profiling and empirically evaluate the overhead and efficacy of our system. Finally, we conclude and discuss our future research plans in Section 5.

2. Background and Related Work

To enable dynamic and adaptive customization of the system stack (all software layers including the application), we require techniques that capture accurate performance data across the system that we can use to guide optimization decisions. Currently, extant approaches to profiling capture only single-layer information, e.g., within a virtual execution environment (JVM or CLR) [2, 9, 5, 17, 11, 21], across the system but agnostic to JVM-process internals [19, 15], employ complex software instrumentation [13, 23, 26], or require hardware extensions [16, 18, 10, 4]. To achieve efficient, full-system profiling currently requires multiple, complex, profiling tools and tedious, inaccurate, offline integration of profile data. To address these limitations, we investigate a full-system profiling system that captures lightweight hardware performance event data across both runtimes for high-level languages, e.g., Java and Microsoft .Net, and for code within the operating system. Our system is called the Vertically Integrated Profiler (VIProf).

VIProf is based on a popular, software-based, operating system profiler called OProfile [19] that we extended to enable the integration of internal JVM profile data. OProfile is a system-wide profiling system that captures hardware performance counter events and correlates them with various parts of the executing system, including applications, libraries, and operating system functions. This system, however, does not support dynamically generated code, such as JIT or dynamically compiled code, and views a JVM instance as a single application without regard as to which applications or programs execute within it. As a result, we are unable to investigate the performance of high-end Java programs, applications servers and their applications, Grid services, and Java middleware in concert with OS performance and OS-JVM interaction. Extant recent approaches to JVM profiling, in particular vertical profiling [8], provide a thorough examination of JVM and Java program internals, but do not capture fine-grain OS activity and the JVM-OS interaction. To address the limitations of these two types of profiling systems, we have developed VIProf. VIProf is able to correlate hardware performance counter events with all code that executes in the system regardless of where it executes (user or kernel space) and how it is generated and linked (dynamically or statically).

An alternative approach to the one that we take with VIProf, is that of the Performance and Event Monitoring (PEM) infrastructure from IBM Research [27]. PEM combines information collected by several distinct monitoring streams that are provided by agents instrumented into the application or operating system (K42), or by native hardware counter agents on the supported hardware (Power Mac G5). VIProf, however, is a lightweight, whole-system, hardware-event profiler that provides a unified perspective of the entire execution stack, and requires no instrumentation or combination of multiple monitored streams. Moreover, VIProf, as a result of our use of OProfile as a base infrastructure, supports different operating systems and hardware platforms – extant profiling system commonly support a single OS or architecture. Finally, our implementation is simple and general enough to support a wide range of virtual execution environments (multiple Java virtual machines as well as Microsoft .Net common language runtimes).

3. Vertically Integrated Profiler (VIProf)

VIProf extends the OProfile to enable integrated profiling across the virtual layers of a system. OProfile consists of a Linux kernel module, and a user level application. The kernel module sets up the hardware performance counters (HPCs) with the user's settings and requests a Non-Maskable Interrupt (NMI) to be raised whenever a configurable threshold value is reached. When the prescribed number of hardware events occur, an HPC overflows and an NMI is raised by the OS. The kernel module handles the interrupt by reading the active Program Counter (PC) value from the processor. OProfile matches the PC value to a virtual memory region, and identifies the corresponding binary or library. Further, OProfile computes the offset into the corresponding object file to pinpoint the method that was executing at the time the interrupt is raised. OProfile adds this information, to which we refer to as a sample, to a buffer for later servicing by a user-level, OProfile daemon. Periodically, this daemon processes the sample buffer and writes the samples to disk. OProfile also includes post-processing utilities to enable flexible reporting.

The primary limitation of OProfile (or any other system-wide profiler) is in profiling dynamically generated code.

```
Time %   Dmiss %  Image name      Symbol name
 9.6393   0.4586  RVM.map         com.ibm.JikesRVM.classloader.VM_NormalMethod.getOsrPrologueLength
 5.4750   0.2293  RVM.map         com.ibm.JikesRVM.classloader.VM_NormalMethod.hasArrayRead
 5.3216   0.2784  RVM.map         com.ibm.JikesRVM.opt.VM_OptCompiledMethod.createCodePatchMaps
 4.9421   1.3429  JIT.App         edu.unm.cs.oal.DaCapo.JavaPostScript.Red.Scanner.Scanner.parseLine
 3.6599   0.1146  RVM.map         com.ibm.JikesRVM.opt.VM_OptGenericGCMapIterator.checkForMissedSpills
 3.6019   0.5077  RVM.map         com.ibm.JikesRVM.MainThread.run
 2.6813  28.2673  libc-2.3.6.so   memset
 1.8861   0.0491  RVM.map         com.ibm.JikesRVM.classloader.VM_NormalMethod.finalizeOsrSpecialization
 1.7619   0       RVM.map         com.ibm.JikesRVM.opt.VM_OptMachineCodeMap.getMethodForMCOffset
 1.5274   0.1802  RVM.map         java.util.Vector.trimToSize
 1.3186  10.0066  libxul.so.0d    (no symbols)
 1.1783   0.1474  libfb.so        fbCopyAreammx
 1.1268   0.0164  libfb.so        fbCompositeSolidMask_nx8x8888mmx


Time %   Dmiss %  Image name                                      Symbol name
69.6552  15.9909  RVM.code.image                                 (no symbols)
 2.3053  39.6149  libc-2.3.6.so                                  memset
 1.9327   0.6569  anon (range:0x65000000-0x65700000),JikesRVM   (no symbols)
 1.2961   0.4304  anon (range:0x65000000-0x65b00000),JikesRVM   (no symbols)
```

Figure 1. Sample profile generated by VIProf (above) and Oprofile (below) for the Dacapo ps benchmark. (Truncated for brevity). The hardware events profiled are `GLOBAL_POWER_EVENTS`**(time) and** `BSQ_CACHE_REFERENCE` **(L2 data cache misses) shown on the first and second columns respectively**

Such code is typical for programs that execute using a virtual machine (VM), e.g., those developed in languages like Java and the Microsoft .Net languages. The format of such programs is an architecture-independent format (to enable portability) that is converted by a VM using an interpreter or dynamic (sometimes referred to as a just-in-time (JIT)) compiler. The location and layout (needed by OProfile to map a PC value to its corresponding binary) of dynamically generated code bodies are thus determined and assigned at runtime and are stored in the VMs virtual memory region. To further complicate matters, virtual machines *re-compile* frequently executing code in an effort to extract higher performance through additional optimization. Thus, the location and layout of code can change dynamically. Finally, if these code bodies are stored in a garbage-collected section of JVM memory, certain garbage collectors may also move code bodies. We refer to dynamically generated code bodies as *JIT code* throughout.

VIProf is a set of OProfile extensions to handle JIT code and VM internals. The extensions enable OProfile to identify a sample as belonging to JIT code and to retrieve information from the VM in which the code is executing. We also map code bodies to its high-level information (methods in an application) so that we can attribute performance data to particular methods. Two key extensions we contributed with include the *Runtime Profiler* and the *VM agent*.

Runtime Profiler. The runtime profiler is the OProfile daemon that runs whenever we wish to log the samples. It is the main source of profiling overhead, extra care must be therefore taken to ensure minimal work is done by this daemon. We extend this daemon by a mechanism that allows a VM to register the fact that it is executing dynamically generated code. The virtual machine also registers the boundaries of its memory heap. Within the daemon, the logging code will consult this information before deciding to log a sample as being anonymous. Instead, if it is found to fall within the boundaries of the VM's heap, the sample will be logged as a JIT.App sample. Apart from a few other limited VM probing routines, this added mechanism is the only extra work that needs to be done at runtime. We evaluate the overhead associated with this daemon in Section 4.

VM Agent. A counterpart to the runtime profiler is the VM agent. This module is responsible for tracking JIT compilations and any GC-induced code body moves. The agent is implemented as a library with several hooks in the VM's code. Specifically, we add instructions in the body of the 'compile' and 'recompile' methods within the VM to log the beginning address, size and signature of the method that was just compiled into a buffer. We also instrument the GC 'move' method within the VM to mark a method's body as having been moved. We simply flag it instead of actually logging it in order to avoid undue overhead. This is because the body of the GC methods are highly tuned and any calls to the outside of their code space will result in a significant performance hit.

At specific points during execution, we process code buffers by writing out a JIT code map to disk. We also traverse a list of known compiled methods and write out the information about any method that was flagged by GC. We then notify the OProfile daemon and request that the written map be associated with the logged JIT.App samples.

3.1. Handling GC-Managed Code

In some VMs, such as the open-source Jikes Research Virtual Machine (Jikes RVM), the code and data regions are both interwound into a single heap; even though the heap may be segregated for optimized garbage collection and memory performance. This presents an additional challenge since the body of a method can exist at several different memory locations during a single execution.

We overcome this challenge by viewing each instance of a Garbage Collection as a cascaded execution epoch. In our Runtime profiler, instead of designating samples to the JIT application, we designate them to a particular execution epoch of the JIT application. At the same time, at the end of each execution epoch, we write out a code map corresponding to the method addresses of this particular epoch. We perform this write just before the launching of the garbage collection. It is important to note that this is a partial write; since it only includes methods that were compiled (or recompiled) since the previous code map write. It also includes the methods that were moved by the previous garbage collection.

During logging time, a sample will be assigned a particular epoch. The post processing tool will attempt to find the corresponding method in that particular code Map. It is possible that the method would not be found, indicating that the sample belongs to a method that was neither compiled nor moved during this particular epoch. In such case, the post processing tool will traverse the set of code maps backwards until it identifies the first occurrence of a method which used to have addresses corresponding to the sample.

3.2. Post Processing Tools

A key to our low overhead implementation of the whole system profiler is that we delay most of the work to the offline profile analysis stage. OProfile comes with a powerful set of post processing tools that are able to categorize, sort and display sample information in a variety of ways. The initial step in the working of these features is the reading and grouping of the sample files.

We modify this component by adding code that will read the map files generated by the VM agent. Since these maps are organized into sequential files - corresponding to the GC epochs of the VM - the tools will initially search for a sample in the map file corresponding to the epoch during which the sample was recorded. If the sample is not found in the epoch's map, the tool will search the immediately preceding map and so on. This guarantees that the method which the sample will be associated with is the most recently compiled - or moved - method to occupy that address space.

The VM itself is an application whose job is to execute the portable object files (e.g. Java Byte Code). Many VMs are written in C or C++ (e.g. Sun Hotspot) and are compiled directly into object files. Since object files are profiled directly by OProfile, no additional work is needed to include the execution information associated with the VM.

However, some VMs, such as the Jikes RVM, are written mostly in Java and as such will not be profiled directly by OProfile. Luckily, the build mechanism for Jikes RVM produces a static image (in a Jikes internal format) and an associated map. We modify the OProfile post processing tool to read in the Jikes RVM internal map and use it to process samples associated with the VM component of the execution. Moreover, there is a small bootstrap application responsible for loading the Jikes RVM class image which is written in C. This application is compiled into an object file and no additional work is needed to profile it.

4. Results

To demonstrate the usefulness and low overhead of VIProf, we used it to profile several benchmarks. In this section we present our experimental methodology, and results of our tests, including a sample profile output and an evaluation of the overhead of our system.

4.1. Methodology

VIProf is an extension to version 0.9.2 of the Oprofile system [19]. The Virtual Machine used is version 2.4.5 of the Jikes Research Virtual Machine [11]. We use a Debian sarge distribution [6] running Linux Kernel 2.6.20.16. All experiments are performed on a single core Intel Pentium 4 Xeon running at 3.4MHz with 2GB of RAM.

We profile three groups of benchmarks: Spec JVM98 [25], Dacapo [3], and Spec Pseudo JBB [24]. JVM98 and Dacapo are collections of short and medium length applications for a variety of tasks including compilers, string manipulation, databases, decoders and other applications designed to model typical Java applications. Pseudo JBB is a variation of Spec JBB which models several warehouses servicing transactions. It is a longer running application than JVM98 and Dacapo. pseudo JBB is configured to have a fixed number of transactions, allowing a direct measure of the execution time. We use an input of '100' for JVM98 benchmarks and 'large' for Dacapo benchmarks. We use 3 warehouses with 100K transactions for pseudo JBB.

For each benchmark, we measure execution time by running the benchmark 10 times, eliminating the fastest and slowest run, and then averaging the remaining 8 runs. We start VIProf just prior to benchmark launch and we configure it to measure the execution time of the benchmarks only.

4.2. Case Study

Figure 1 shows the output of VIProf (the upper portion) versus the output of Oprofile (the lower portion) for an identical run of the Dacapo ps benchmark. In the Oprofile output, the Java Application and Virtual Machine are both profiled as black boxes. However, we are able, using VIProf to identify the systems execution patterns and view all methods side by side. In addition to relative method weights

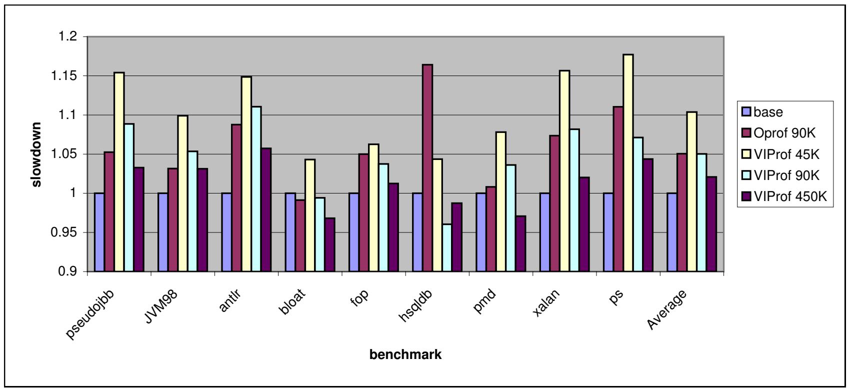

Figure 2. Overhead of profiling with VIProf compared to Oprofile. Higher bars indicates slower relative execution time.

and correlated hardware events, VIProf also extends the call graph functionality of Oprofile to include call sequence profiles across layers. We omit these results for brevity.

VIProf output demonstrates the utility and convenience of our profiler for whole system profiling. This level of detail profiling is indispensable to our overall goal of vertically integrated optimization.

4.3. Overhead

Figure 2 shows the execution overhead associated with VIProf at 3 different profiling frequencies. The figure also shows the Oprofile overhead for the median frequency (90K). The execution times are normalized to base execution time (i.e. no profiling or running VM agents). Figure 3 shows the base execution time of the evaluated benchmarks for reference.

On average, VIProf adds negligible overhead to what Oprofile already introduces. While most benchmarks experienced a slight slowdown compared to Oprofile, a few experienced speedups. We believe this is due to VIProf avoiding the anonymous memory logging code in Oprofile (which we replace with our VIProf mapping code). A few benchmarks exhibited speedups as compared to the base execution time. We found these to be medium lengthed Dacapo applications (`hsqldb, bloat`). We believe this is due to system noise and the uncertainty involved in full system measurements.

For a moderate level of sampling (1 in 90K cycles), Oprofile generally slows down the system by an average of 5%. Our system also exhibits a similar slowdown on average. This compares favorably to other similar profilers such as Vertical Profiling which reported 7% average profiling overhead (although vertical profiling is limited to covering the VM and application domain only).

For a median sampling frequency (90K) with VIProf, the majority of benchmarks experienced slowdowns less than 10%, with one exception, `antlr`, recording a slowdown above 10%. Four benchmarks had slowdowns that were less than 5%. Longer running benchmarks generally experienced the smaller slowdowns, due to the amortization of the cost of writing out the code maps. Further, as the code reaches higher optimization levels and the GC moves these regions to the mature space, there is less need for any runtime work to be done to support our VIProf system.

Benchmark	Base time
pseudoJBB	31
JVM98 (average)	5.74
antlr	8.7
bloat	28.5
fop	3.2
hsqldb	43
pmd	16.3
xalan	137.9
pseudoJBB	22.2
Average	32.9

Figure 3. Base execution time in seconds for the benchmarks.

5. Conclusions and Future Work

Investigating optimization and specialization opportunities across the entire system stack is the main focus of our research. In this paper, we present VIProf: an efficient profiling system that captures performance transparently across the entire system stack. VIProf is a first step towards interrelating performance bottlenecks across layers (OS, VM, and application), and identifying potential optimization and specialization opportunities for the entire execution stack. We show that VIProf captures interesting events across the system with very low overhead.

As part of future work, we plan to integrate Xen virtualization extensions into VIProf to integrate profiling of the Xen layer (via XenoProf [29]) as well as multiple concurrently executing software stacks. In addition, we plan to investigate profile-guided optimizations across multiple layers of the execution stack. In particular, we are interested in customizing the VM and OS and their interaction according to the changing behavior of an application and the available resources.

References

[1] A. Whitaker and M. Shaw and S. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[2] M. Arnold and B. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2001.

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2006.

[4] C. Zilles and G. Sohi. A Programmable Co-processor for Profiling. In *High Performance Computer Architecture(HPCA)*, pages 241–253, 2001.

[5] T. Chilimbi and M. Hauswirth. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2004.

[6] Debian gnu/linux. http://www.debian.org.

[7] S. W. Galley. PDP-10 virtual machines. In *Proceedings of the workshop on virtual computer systems*, pages 30–34, New York, NY, USA, 1973. ACM Press.

[8] M. Hauswirth, A. Diwan, P. F. Sweeney, and M. Moze. Automating vertical profiling. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2005.

[9] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

[10] J. Dean and J. Hicks and C. Waldspurger and W. Weihl and G. Chrysos. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *ACM/IEEE MICRO*, pages 292–302, 1997.

[11] JikesRVM IBM research publications. `http://www-124.ibm.com/developerworks/oss/jikesrvm/info/pubs.shtml`.

[12] C. Krintz and R. Wolski. Using phase behavior in scientific application to guide linux operating system customization. In *Workshop on Next Generation Software at IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2005.

[13] M. Arnold and B. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 168–179, 2001.

[14] S. E. Madnick and J. J. Donovan. Application and analysis of the virtual machine approach to information system security and isolation. In *Proceedings of the workshop on virtual computer systems*, pages 210–224, New York, NY, USA, 1973. ACM Press.

[15] A. V. Mirgorodskiy and B. P. Miller. Crosswalk: A tool for performance profiling across the user-kernel boundary. In *International Conference on Parallel Computing (PARCO)*, pages 745–752, 2003.

[16] H. Mousa and C. Krintz. Hps: Hybrid profiling support. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 38–50, 2005.

[17] Microsoft phoenix framework. `http://research.microsoft.com/phoenix`.

[18] P. Nagpurkar, H. Mousa, C. Krintz, and T. Sherwood. Efficient remote profiling for resource-constrained devices. *ACM Trans. Archit. Code Optim. (TACO)*, 3(1):35–66, 2006.

[19] Oprofile. `http://oprofile.sourceforge.net`.

[20] P. Barham and B. Dragovic and K. Fraser and S. Hand and T. Harris and A. Ho and R. Neu gebauer. Virtual machine monitors: Xen and the art of virtualization. In *Symposium on Operating System Principles*, 2003.

[21] M. Paleczny, C. Vick, and C. Click. The Java HotSpot(TM) Server Compiler. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*, Apr. 2001.

[22] R.A. Meyer and L.H. Seawright. A Virtual Machine Time Sharing System. In *IBM Systems Journal*, 1970.

[23] S. Sastry, R. Bodík, and J. Smith. Rapid Profiling via Stratified Sampling. In *Annual International Symposium on Computer Architecture (ISCA)*, pages 278–289, July 2001.

[24] SpecJBB Benchmarks. `http://www.spec.org/jbb2000`.

[25] SpecJVM'98 Benchmarks. `http://www.spec.org/osg/jvm98`.

[26] T. Heil and J. Smith. Relational Profiling: Enabling Thread-Level Parallelism in Virtual Machines. In *ACM/IEEE MICRO*, pages 281–290, 2000.

[27] R. W. Wisniewski, P. F. Sweeney, K. Sudeep, M. Hauswirth, E. Duesterwald, C. Cascaval, and R. Azimi. Performance and environment monitoring for whole-system characterization and optimization. In *IBM PAC2 Conference*, Oct. 2004.

[28] J. Xenidis. rHype: IBM Research Hypervisor. In *IBM Research*, March 2005.

[29] Xenoprof. `http://xenoprof.sourceforge.net`.

[30] L. Youseff, R. Wolski, B. Gorda, and C. Krintz. Evaluating the Performance Impact of Xen on MPI and Process Execution for HPC Systems. In *International Workshop on Virtualization Technology in Distributed Computing (VTDC) w/SC06*, Nov. 2006.

[31] L. Youseff, R. Wolski, B. Gorda, and C. Krintz. Paravirtualization for HPC Systems. In *ISPA Workshops*, volume 4331 of *Lecture Notes in Computer Science*, pages 474–486. Springer, 2006.

[32] L. Youseff, R. Wolski, and C. Krintz. Linux Kernel Specialization for Scientific Application Performance. Technical Report 2005-29, UCSB, Nov 2005.