ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay

Min Xu Vyacheslav Malyugin Jeffrey Sheldon Ganesh Venkitachalam Boris Weissman
VMware Inc.

Abstract

Execution trace is an important tool in computer architecture research. Unfortunately, existing trace collection techniques are often slow (due to software tracing overheads) or expensive (due to special tracing hardware requirements). Regardless of the method of collection, detailed trace files are generally large and inconvenient to store and share.

We present ReTrace, a trace collection tool based on the deterministic replay technology of the VMware hypervisor. ReTrace operates in two stages: capturing and expansion. ReTrace capturing accumulates the minimal amount of information necessary to later recreate a more detailed execution trace. It captures (records) only non-deterministic events resulting in low time and space overheads (as low as 5% run-time overhead, as low as 0.5 byte per thousand instructions log growth rate) on supported platforms. ReTrace expansion uses the information collected by the capturing stage to generate a complete and accurate execution trace without any data loss or distortion. ReTrace is an experimental feature of VMware Workstation 6.0 currently available in Windows and Linux flavors for commodity IA32 platforms. No special tracing hardware is required.

We have three key results. First, we find that trace collection can be done both efficiently and inexpensively. Second, deterministic replay is an effective technique for compressing large trace files. Third, performing the trace collection at the hypervisor layer is minimally invasive to the collected trace while enabling tracing of the entire system (user/supervisor level, CPU, peripheral devices).

ReTrace is a rapidly evolving technology. We would like to use this paper to solicit feedback on the applicability of ReTrace in computer architecture research to help us refine our future development plans.

1 Introduction

When computer architects want to peek into a running system, they often use trace-driven techniques, which means collecting run time information (herein called execution traces) from a workload. Offline, the execution traces can be analyzed in more detail, such as performing cache simulations. Execution traces vary from sparse traces (e.g., collecting context switches) to detailed traces (e.g., collecting memory references). Not surprisingly, it is the detailed execution traces that are the most challenging to collect.

In their survey, Uhlig and Mudge noted the following challenges in trace collection [25].

- Maximizing trace completeness and the detail level. For example, a memory reference trace is more complete if it includes both operating system (OS) and application memory references. Similarly, a memory reference trace is more detailed if it includes the annotation of virtual address to physical address mapping for each reference.
- Minimizing trace distortion. In other words, the trace collected should faithfully represent the program being traced. For example, trace collection itself should not introduce extra memory references in a memory trace. Perhaps more subtly, tracing should minimize time dilation and memory dilation, which occurs when tracing causes a program to run slower (from its own perspective) or to consume more memory (in its own address space).
- Reducing trace file size. The trace collecting tool should produce highly compact trace files, which are easier to store and share with other researchers. Compact trace files also enable long execution traces to be collected for different analysis needs.
- Fast, inexpensive and easy to operate.

To the best of our knowledge, none of the existing trace collection techniques sufficiently solves all these challenges. We briefly survey these techniques in Section 6. This paper presents a deterministic replay based execution trace technique, ReTrace, which better solves all these challenges. ReTrace decomposes traditional trace collection process into two steps: *ReTrace capturing* and

ReTrace expansion. This decomposition is the key to meet the four challenges above. During ReTrace capturing, minimum information on execution nondeterminism is logged. This reduces the necessary trace file size and trace distortion, because the capturing overhead is minimal. During ReTrace expansion, maximum information on all aspects of the execution can be collected. The resulting trace is complete and detailed without added trace distortion. By building on virtual machine technology, ReTrace is fast, inexpensive and easy to operate.

Deterministic replay is the key technology that enables the decomposition of ReTrace capturing and ReTrace expansion. Our deterministic replayer is built on top of VMware's virtual machine monitor [1, 24]. A virtual machine monitor enables multiple operating systems (guests) to share the same physical hardware (host) [21]. Our deterministic replayer can record and replay the guest execution (including both guest OS and guest applications) at the IA32 Instruction Set Architecture (ISA) layer. During recording, only minimal information to capture the nondeterminism is recorded, thereby, minimizing the trace distortion and trace file size. Because of the low overhead in recording, ReTrace is accurate, fast and easy to operate. During replaying, complete and detailed execution traces can be selectively collected in batch mode. Even though replaying is much slower than recording, no additional trace distortion is introduced because of deterministic replay. Finally, ReTrace is implemented completely in software and is available as an experimental feature in VMware Workstation 6.0[1] without requiring special tracing hardware.

The rest of the paper is organized as follows. Section 2 briefly introduces our deterministic replayer, which is the key enabling technology of ReTrace. Section 3 and Section 4 describe in detail the two steps of using ReTrace – ReTrace capturing and ReTrace expansion. Section 5 presents evaluation results of ReTrace using SPEC CPU2006 [12] and Apache Benchmark [2]. Finally, we briefly discuss existing work in Section 6 and conclude in Section 7.

2 Deterministic replay of virtual machines

Virtualization allows multiple operating systems (OS) to simultaneously execute on a single physical hardware platform [21]. Virtualization provides many benefits, such as server consolidation and security isolation, to computer users. A Virtual Machine (VM) consists of necessary state, including the CPU, memory and I/O devices, to run a guest OS. The Virtual Machine Monitor (VMM) is a thin layer of software that sits between the physical hardware and the guest OS to enable sharing of the physical CPU, memory and I/O devices among multiple VM. Until recently, the IA32 architecture has not permitted the classical trap-and-emulate virtualization. Adams and Agesen present an up to date study on virtualizing IA32 [1].

Deterministic replay [3, 10, 17, 18, 20, 23, 28] creates an execution that is logically equivalent to an original execution of interest. Two executions are logically equivalent if they contain the same set of dynamic instructions, each dynamic instruction computes the same result in the two executions, and the two executions compute the same final state. A deterministic replayer is a software or hardware component that records an execution and replays it deterministically. Our deterministic replayer is based on VMware's VMM [1, 24], which is an ideal place to record and replay the guest execution as we will show. Our replayer supports full-system recording and replay, i.e., the entire VM execution, including guest OS and guest applications, is recorded and replayed.

During recording, all sources of nondeterminism from outside the virtual machine are captured and logged in a log file. These include data and timing of inputs to all devices, including virtual disks, virtual network interface cards (NIC), *etc.* A combination of techniques, such as device emulation and binary translation, are used to ensure deterministically replay as long as the recorded device input data are replayed at right time.

We solve two challenging problems in this replay system. First, we efficiently keep track of number of instructions executed in the recorded execution, so that we can reinject the same nondeterministic events at the same execution points during replay. Second, we ensure the VM is deterministic, even though underlying real machine is nondeterministic.

Our deterministic replayer is currently limited to uniprocessor VMs and it supports popular IA32 platforms, such as Pentium 4, Opteron and Core 2[2].

3 ReTrace capturing

The first step in using ReTrace is trace capturing. The resulting log file is called the replay log. The replay log contains multiple types of log entries. The log entries are sequentially stored in the execution order. Some entries records the nondeterministic input values needed during replay. Others records both timing and values of nondeterministic events.

[1] VMware Workstation 6.0 is available to researchers through VMAP [13].

[2] Opteron and Core 2 are supported without hardware acceleration in the released VMware Workstation 6.0 software.

To use ReTrace, users need to execute the target workload in a virtual machine environment. The virtual machine environment is almost identical to its native counterpart. Therefore, no porting of the workload is necessary. Then, the user needs to turn on ReTrace recorder to capture the replay log while the target workload is executing. We now discuss the following two questions in more detail.

1. How much trace distortion do the virtual machine and ReTrace recorder incur?
2. How usable is the ReTrace Recorder in both workload setup and gathering the replay log?

3.1 Trace distortion

Question 1 is important for ReTrace because the capturing process is the only source of trace distortion in ReTrace. The second step, ReTrace expansion, is a deterministic replay, which by definition is logically equivalent to the captured execution. Thus it does not incur additional distortion[3].

Much of the trace distortion come from the virtualization layer, where virtual devices may have different characteristics than the real devices. For instance, a virtual keyboard command that finishes in one instruction in a VM will take much longer to finish with a real keyboard. Therefore, while a native execution trace contains thousands of instructions between the start and finished of the command, the virtual execution trace will contain just one instruction between the two events. However, the extent of this kind of trace distortion is limited, because the virtual machine monitor strives to implement faithful device models to support unmodified guest OS. It would also be possible to plug in more detailed device models in the future.

Secondly, the trace capturing overhead incurs minimal trace distortion, because (1) the virtual machine layer executes guest instructions at close to native speed and (2) only the rare nondeterministic events are intercepted and logged by the recorder.

3.2 Usability

The capturing process is as easy as pushing a recording button if the target workload is already imported into a virtual machine. Otherwise, tools like VMware Converter [27] make it is easy to import workload into a virtual machine. During capturing, there is minimal impact on the interactive responsiveness of a workload, due to the low run-time overhead of trace capturing. This eases workload setup, when interactive responsiveness is important.

4 ReTrace expansion

After ReTrace capturing, users can obtain more detailed execution traces by deterministically replaying the replay log with additional probes added to the replayed execution. We call this step ReTrace expansion. The resulting execution trace is called the *full trace*. Thanks to the portability of virtual machines and the small size of the replay log, ReTrace expansion can be moved to a different computer[4] to facilitate sharing of execution traces among researchers.

ReTrace expansion can generate different types of execution traces, such as instruction traces, memory reference traces and device event traces. All these can be done without incurring any trace distortion because of deterministic replay. The full trace can be extremely verbose. For example, memory reference trace may include both virtual and physical addresses, as well as details of virtual-to-physical mappings. In addition, it should be possible to directly attach a timing simulator to ReTrace during ReTrace expansion. This avoids generating and storing large full trace files. Finally, it would be possible to filter the full trace to collect traces only for a given user-level application in future releases of ReTrace.

The speed of the deterministic replay depends on several factors. Without trace expansion, the replay speed can be *faster* than the original execution, because replay skips over the idle time of HLT instructions [14][5] in the original execution. With trace expansion, the replay speed is much slower than the original execution, depending on the detail level of the expanded execution trace. Fortunately, the deterministic replay is very easily parallelizable. During replay (or recording), one can take multiple VM checkpoints and subsequently replay can start from those checkpoints. Therefore, even though ReTrace expansion is a slower process than ReTrace capturing, a full trace can be potentially generated quickly using multiple machines expanding the replay log in parallel.

ReTrace also supports *selective* trace expansion. In particular, a keyboard command controls when to start and stop the trace expansion during deterministic replay. In

[3] We assume that users are only interested in an ISA level execution trace, which does not contain ISA-invisible execution information, such as cache misses or speculative instruction execution serviced by underlying hardware at trace collection time.

[4] Restrictions on processor types and software versions apply to this portability feature.

[5] HLT stops the processor execution until the next interrupt. The time to replay HLT can be much shorter than its original execution.

other words, while watching the fast replay of the captured execution, a user can select the portions of execution that she is the most interested to expand. If the user activates ReTrace expansion during replay, the virtual machine monitor enters an instruction interpretation mode. In this mode, before and after executing each instruction, the VM state is inspected and logged in a separate full trace file. The full trace file is compressed with gzip [29]. In current version of ReTrace, the execution trace is in plain text, which gives users flexibility in how to use the execution trace with their simulation tools. A trace file contains the following fields.

- Processor CPL
- Instruction Pointer (IP)
- Exceptions, Faults and Interrupts
- Registers: EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, EFLAGS
- Segments: ES, CS, SS, DS, FS, GS
- Control registers: CR0, CR2, CR3, CR4

Currently, the trace content is far from complete and the plain text format does not conform to any widely accepted standard. We are looking forward to hearing from our users on which other types of information should be added to the trace and which standard trace format should ReTrace use.

5 Experimental results

We first describe the experimental methodology, then the runtime impact and log sizes of ReTrace.

5.1 Methodology

We evaluate ReTrace with two foci: its performance impact and the replay log size and the full trace size. We use two types of workloads: SPEC CPU2006 [12] and Apache Benchmark (AB) [2]. All experiments are conducted within a Linux VM.

The VM has one virtual CPU, 2GB of memory and a 40GB virtual SCSI disk. The guest OS is Xubuntu 6.06 running an unmodified Linux 2.6 kernel packaged with the distribution. The host OS is Ubuntu 6.06 also running Linux 2.6. The host system has two dual core AMD Opteron CPUs with 4GB of memory and a 120 GB SATA disk.

The SPEC CPU2006 benchmark suite consists of 12 integer benchmarks and 17 floating point benchmarks. We compile the benchmarks using a gcc 4.2 snapshot from March 16, 2007. We run each benchmark 5 times with training inputs. The entire experiment takes about 14 hours.

Despite the non-trivial setup process of SPEC CPU2006, ReTrace does not make it harder than on a native machine. For example, correct configurations have to be selected to successfully compile all benchmarks as well as gcc itself. The entire setup process is no worse than on a real machine because of the low overhead incurred by virtualization.

The Apache benchmark is a simple static web server benchmark. A driver program repeatedly fetches static web pages from an Apache 2.0 web server. Both the driver and the web server run in the same VM. We intend to use this benchmark to test ReTrace under OS-intensive and IO-intensive workload. Like other commercial workloads, the performance is measured by throughput (KB/s) not latency. We perform five 30-second runs for each configuration.

5.2 Run-time impacts

Figure 1 shows the run-time impact of ReTrace capturing. We compare the run-time in the VM without and with ReTrace capturing. Each benchmark is run 5 times and the normalized average run-time is plotted. The rightmost pair of bars shows the geometric mean of the run-time overheads. The run-time overhead of ReTrace capturing range from 0.7% to 31% with a geometric mean of 5.09%. This shows that ReTrace has extremely low overhead for CPU intensive workloads[6].

ReTrace capturing incurs significantly more overhead (2.6x) on this OS and IO intensive workload than the CPU intensive workloads. Apache benchmark reports a throughput of 747 KB/s without ReTrace but only 283 KB/s with ReTrace. The overhead primarily come from unoptimized paths in the VMM during guest kernel execution. We understand that the problem is solvable and we are working on a fix for the problem. Even at a slowdown of 2.6x, ReTrace capturing has much lower overhead than existing tracing methods, which often incur at least 1 or 2 orders of magnitude slowdown [5, 19].

We do not compare ReTrace's performance with native (non-virtualized) executions for two reasons. First, virtualization is being widely accepted as a standard way of deploying server workloads. Therefore, execution traces

[6]By default, the released version of ReTrace has slightly more overhead due to increased runtime checks, which help us catch bugs in the field.

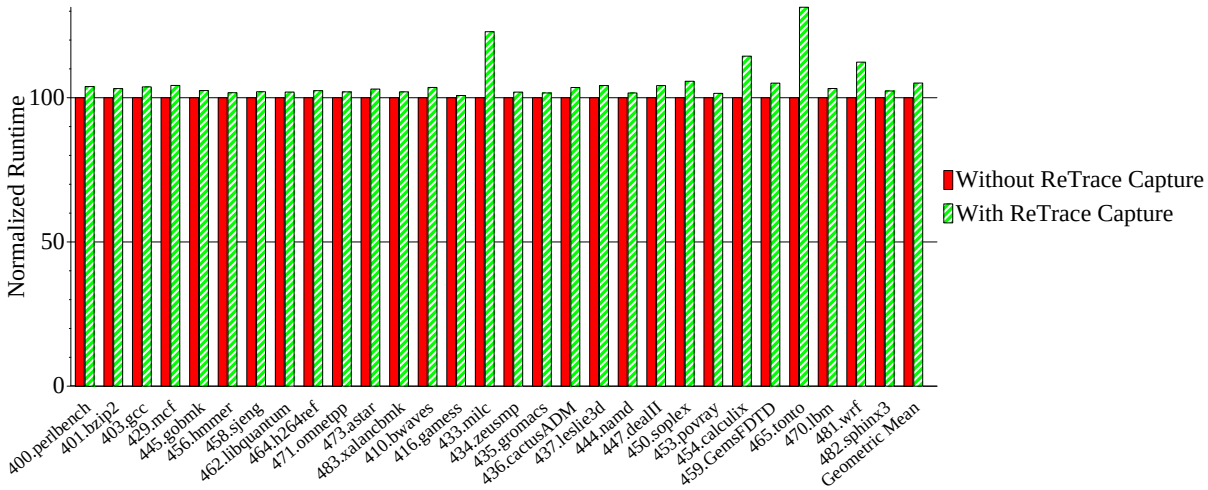

Figure 1: SPEC CPU2006 INT and FP performance without and with ReTrace capturing. Each benchmark is run 5 times and the average runtime is compared. The geometric mean is 5.09% slowdown with ReTrace capturing.

collected within a VM are relevant to computer architecture research. Second, virtualization usually incurs acceptable overhead in production environments. Therefore, execution traces collected within a VM should be close to the native executions.

Table 1 shows the run-time of deterministic replay without and with ReTrace Expansion. Due to the slow interpretation mode and the detailed trace collection, ReTrace expansion is at least 2 orders of magnitude slower than an non-expansion replay. However, this slow expansion speed does not incur any trace distortion (because of deterministic replay) and is unlikely to cause usability problems (because it does not require any user interaction). In fact, we stopped the expansion after 2 hours, because the huge size of the resulting execution trace files quickly become the bottleneck. It is more likely that users will use the selective trace expansion feature to expand only small and interesting segments from the much longer replay log. In other words, a user can repeatedly "zoom in" onto interested execution segments using deterministic replay.

5.3 Replay log and full trace size

Table 2 shows that the captured log is *4 orders of magnitude* smaller than the expanded log file for 400.perlbench and Apache Benchmark. This is not surprising because ReTrace records only those non-deterministic inputs, not a full instruction trace. In addition, because ReTrace replays the entire virtual machine, rather than only a CPU, ReTrace avoids logging CPU inputs from devices (such as inputs from the disk controller). On average, ReTrace generates 4.8 byte per thousand instructions. In contrast, a previous CPU-centric deterministic replayer generates log file at approximately 14 bytes/kilo-instruction (70 instructions per byte) [22], partly because it logs CPU inputs.

Table 3 shows the same log size comparison after applying gzip compression, which reduces the replay log by another 5X to 10X. The lower compression ratio for 400.perlbench is due to more idle time in the start-up of the workload. More idle time results in (relatively) more nondeterministic timer events per instruction. The timer events tend to be less compressible. In other words, more idle time results in more log data per instruction, but less log data *per second*.

In practice, we observe a complete Windows XP boot-shutdown generates 776 KB of compressed replay log.

6 Related work

Uhlig and Mudge presented a comprehensive survey on trace-driven memory simulation [25]. In this section, we briefly contrast ReTrace with other trace collection and compression methods, especially those methods developed after Uhlig's survey.

6.1 Trace collection

Trace collection methods can be broadly divided into software-based and hardware-based. Existing software-based trace collection methods incur significant run-time overhead due to software instrumentation cost. To reduce this cost, static program analysis [4, 26] can be applied to minimize the instrumentation points in a program. In contrast, ReTrace works on program binaries without requiring any static program analysis.

	Without Expansion	With Expansion
400.perlbench	117 seconds	> 2 hours
Apache Benchmark	30 seconds	> 2 hours

Table 1: ReTrace expansion speed of 400.perlbench and Apache Benchmark.

	Replay Log	Full Trace	Expansion Factor
400.perlbench	4.1	296,849	72402X
Apache Benchmark	5.4	301,950	55916X

Table 2: ReTrace log size of 400.perlbench and Apache Benchmark. All log size numbers are in Byte/Kilo-instructions.

	Replay Log	Full Trace	Expansion Factor
400.perlbench	1.1	10,047	9134X
Apache Benchmark	0.5	10,366	20732X

Table 3: Gzip compressed ReTrace log size of 400.perlbench and Apache Benchmark. All log size numbers are in Byte/Kilo-instructions.

Some hardware-based methods require expensive and system-specific hardware probing devices, such as Tektronix logic analyzer [11] and hardware bus monitor [9, 6]. A serious problem is that not all traced program activities are visible to the probing device due to effects such as on-chip caching *etc.* Some researchers have proposed instrumenting the traced program to bypass the caching effects. Others utilize on-chip trace collecting hardware in commodity processors [15]. Both on-chip or off-chip hardware tracing methods suffer from the limited size of the hardware trace buffer. When the buffer fills up, the traced program has to be stalled or a discontinued trace will be collected. For example, ITrace, an open source instruction trace facility, can incur 100x slowdown to traced program [19].

6.2 Trace compression

Detailed execution trace are often huge in size. For example, a SPEC CPU2000 perlbmk trace is 601.4 MB, which contains only 214.7 million memory references (a fraction of a second execution) [8]. Trace compression is often needed. For example, Kaplan *et al.* proposed lossy and loss-less compression techniques for memory reference traces [16]. Burtscher proposed a value prediction based method to compress instruction traces [7]. These methods share the common feature that trace compression is applied *after* large trace files have been generated. In contrast, ReTrace captures minimal log file in the first place and the ReTrace compression is independent of what kind of trace analysis is later applied to the trace, *e.g.*, ReTrace does not assume LRU based simulation on the memory reference trace or value predictability.

6.3 Replay-based trace collection

Most recently, deterministic replay based trace collection has been recognized as a promising method.

CITCAT employs a simulator-based deterministic replayer [22]. The key elements of CITCAT is its cache-filtered address trace recorder on a real machine. In contrast with ReTrace, CITCAT differs in two ways. CITCAT needs to modify traced OS to make key non-deterministic events visible on the off-chip system bus. CITCAT is CPU-centric, which does not replay devices.

Bhansali *et al.* from Microsoft presented a framework for instruction tracing based on deterministic replay [5]. Their technique is based on binary translation. Like ReTrace, no special tracing hardware is required. However, the 5X to 17X run-time overhead is much higher than that of ReTrace. Also, unlike ReTrace's full system replay, their replay is at the application level.

In both CITCAT and Bhansali's replayer, the trace log size is larger than that of ReTrace because more run-time information (such as memory reads) are treated as non-deterministic. CITCAT has a log size growth rate from 14 bytes/kilo-instruction and Bhansali's replayer generates log at a rate from 10 to 137 bytes/kilo-instruction.

7 Conclusion

In conclusion, ReTrace is a trace collection tool based on VMware's virtual machine monitor and deterministic replay. ReTrace has extremely low run-time overhead and high trace file compression ratio. We show that deterministic replay can enable efficient and inexpensive trace

collection. Our experiences suggest that virtual machine monitor is an ideal place to collection full-system execution trace, which can be stored as small deterministic log files and can be selectively expanded into a detailed full-system execution trace during the fast replay.

Future work includes extending ReTrace to work with other trace process and simulation tools. Community feedback would be invaluable to ReTrace's future development plan.

8 Acknowledgment

This paper would not have existed without our colleagues on the deterministic replay project at VMware. They are Petr Vandrovec, Dan Scales, Mike Nelson and Eric Lowe. We also thank the virtual machine monitor group at VMware for creating a fantastic environment. We thank Beng-Hong Lim for encouraging us to take on the task of writing this paper. We thank Mark Hill, Ole Agesen, Carl Waldspurger, E Christopher Lewis, Ravi Mummidi and MoBS reviewers for extremely useful feedback.

References

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2006. ACM Press.

[2] Apache Software Foundation. http://www.apache.org/.

[3] David F. Bacon and Seth Copen Goldstein. Hardware-Assisted Replay of Multiprocessor Programs. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, pages 194–206, 1991.

[4] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, 1994.

[5] Sanjay Bhansali, Wen-Ke Chen, Stuart De Jong, Andrew Edwards, and Milenko Drinic. Framework for Instruction-level Tracing and Analysis of Programs. In *Second International Conference on Virtual Execution Environments*, 2006.

[6] P. Bosch, A. Carloganu, and Daniel Etiemble. Complete x86 instruction trace generation from hardware bus collect. In *EUROMICRO*, pages 402–408, 1997.

[7] Martin Burtscher. VPC3: A Fast and Effective Trace-Compression Algorithm. In *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 167–176, 2004.

[8] BYU Trace Distribution Center. http://tds.cs.byu.edu/tds/.

[9] R. Daigle, C. Xia, and J. Torrellas. Low Perturbation Address Trace Collection for Operating System. Technical report, Center for Supercomputing Research and Development, University of Illinois at Urbana Champaign, Mar 1996.

[10] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[11] J. Kelly Flanagan, Brent E. Nelson, James K Archibald, and Knut Grimsrud. BACH: BYU Address Collection Hardware, The Collection of Complete Traces. In *Proc. of the 6th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 128–137, 1992.

[12] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[13] VMware Inc. http://www.vmware.com/partners/academic/.

[14] Intel corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, November.

[15] Maynard Johnson, Scott Jones, John Kacur, Frank Levine, Milena Milenkovic, and Enio Pineda. http://perfinsp.sourceforge.net/itrace.html.

[16] S. Kaplan, Y. Smaragdakis, and P. Wilson. Flexible reference trace reduction for VM simulations. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 13(1):1–38, Jan 2003.

[17] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 Annual USENIX Technical Conference*, April 2005.

[18] Thomas J. Leblanc and John M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, Apr 1987.

[19] Frank Levine. Personal Communication on itrace overhead, Mar 2007.

[20] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording Shared Memory Dependencies Using Strata. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 229–240, Oct 2006.

[21] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[22] Charlton D. Rose. CITCAT: Constructing Instruction Traces From Cache-Filtered Address Traces. Master's thesis, Brigham Young University, 1999.

[23] Mark Russinovich and Bryce Cogswell. Replay for Concurrent Non-Deterministic Shared-Memory Applications. In *Proceedings of the SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 258–266, May 1996.

[24] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

[25] Richard A. Uhlig and Trevor N. Mudge. Trace-Driven Memory Simulation: A Survey. *ACM Computing Surveys*, 29(2):128–170, 1997.

[26] UW-Madison. http://www.cs.wisc.edu/ larus/qpt.html.

[27] VMware Inc. http://www.vmware.com/products/converter/.

[28] Min Xu, Rastislav Bodik, and Mark D. Hill. A Hardware Memory Race Recorder for Deterministic Replay. *IEEE Micro*, 27(1), Jan/Feb 2007.

[29] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.