Virtual Execution Environments: Support and Tools

Apala Guha†, Jason D. Hiser†, Naveen Kumar‡, Jing Yang†, Min Zhao‡, Shukang Zhou†
Bruce R. Childers‡, Jack W. Davidson†, Kim Hazelwood†, Mary Lou Soffa†
‡Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260
†Department of Computer Science, University of Virginia, Charlottesville, VA 22904
{jwd,hazelwood,soffa}@cs.virginia.edu {childers,kumar,zhao}@cs.pitt.edu

Abstract

In today's dynamic computing environments, the available resources and even underlying computation engine can change during the execution of a program. Additionally, current trends in software development favor the flexibility and cost-effectiveness of dynamically loaded components and libraries. Because of these trends, there has been increased research interest in virtual execution environments (VEEs) for delivering adaptable software suitable for today's rapidly changing, heterogeneous computing environments. In this project, we have been investigating tools and techniques to support implementation of VEEs using software dynamic translation (SDT). This paper highlights some of our recent results. One significant result is that we have developed novel translation techniques that reduce the memory and runtime overhead of SDT to negligible levels. We have also developed innovative debugging and instrumentation tools for SDT-based software environments. Together, these results make SDT-based systems viable for solving a wide range of pressing problems. The paper concludes with a discussion of how SDT may offer a solution to one such problem—inherent process variation in emerging chip multiprocessors.

1. Introduction

Over the last decade interest in virtual execution environments (VEEs) has been growing with the increased recognition of their usefulness and power. A VEE provides a self-contained operating environment that facilitates programmatic modification of an executing program for diverse purposes, such as architecture-portability [4, 5], performance [2, 15, 1], instrumentation [19, 17, 16], security [11, 14, 21], and power consumption [6]. Many VEEs execute applications using software dynamic translation (SDT), which has the potential to produce high-quality code and utilize resources efficiently. SDT systems virtualize aspects of the host execution environment by interposing a layer of software between program and CPU. This software layer mediates program execution by dynamically examining and translating a program's instructions before they are executed on the host CPU.

This project has been studying various aspects of SDTs and developing techniques to improve their efficiencies in both runtime performance and memory utilization and to develop tools that will enable the widespread acceptance of SDTs. In this paper, we summarize our recent contributions and discuss future work.

Section 2 briefly describes Strata and the techniques that we developed to reduce the memory requirements and improve performance. Section 2 describes Dimension, a tool that provides instrumentation services for SDTs. A critical issue for software developers is how to debug code running under the control of an SDT system. In Section 4, we describe DeDoc, a framework that allows debugging at the source level for programs that have been transformed by a traced-based dynamic binary optimizer. Section 5 discusses related work and Section 6 presents conclusions and future research directions.

2. Strata Overview

Our first task was to design and develop a SDT that could be used in experimental studies of the issues in building and using SDTs. Figure 1 shows the high-level architecture of Strata, our SDT infrastructure. Strata provides a set of retargetable, extensible, SDT services. These services include memory management, fragment cache management, application context management, dynamic linker, and a fetch/decode/translate engine.

2.1 Performance and Memory

Because VEEs allow programs to be modified as they are running, the overhead of monitoring and modifying a running program's instructions is often substantial. As a result, SDT can be slow, especially SDT systems that are not carefully designed and implemented. Furthermore, SDT systems can increase memory utilization which may be problematic for embedded systems. We investigated several overhead reduction techniques, including indirect branch translation caching, fast returns, and static trace formation, that can improve SDT performance. We also developed techniques to reduce the memory demands of a SDT through the management of exit stubs.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

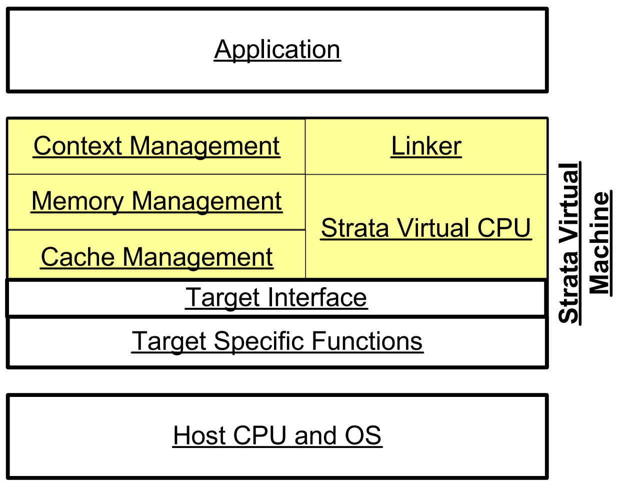

Figure 1: Strata high-level overview.

2.2 Performance

Reducing the overhead of software dynamic translation (SDT) is critical for making SDT systems practical for use in production environments. Using the SPECint2K benchmarks, we performed detailed measurements to determine major sources of SDT overhead. Our measurements revealed that indirect branches were a significant source of overhead. To reduce the number of context switches caused by indirect branches, we used an indirect branch translation cache. This cache maps indirect branch-target addresses to their fragment cache location. With a small 512-entry cache, the overall slowdown was further reduced from an average 4.1x to an average of 1.7x.

To reduce overheads further, we developed a technique to better handle indirect branches that were generated because of return statements. For function returns where the fragment cache holds the return address, function returns can be rewritten to return directly to the fragment cache return address, thereby avoiding a context switch. This technique reduced the SDT slowdown to an average of 1.3x.

Finally, we investigated the usefulness of determining instruction traces statically and using this information to reduce the number of context switches and improving instruction cache locality. This technique resulted in a performance improvement of up to 39% (average 15%) over fragment linking. These results demonstrate that static information can be successfully used to guide SDT and reduce its overhead.

Our results indicate that by applying the techniques described here along with some dataflow analysis of the executable, it may be possible to eliminate SDT overhead entirely [10, 20]. If achieved, this would make SDT a powerful tool for helping software developers achieve a variety of important goals including better security, portability, and better performance.

2.3 Memory

SDTs introduce an extra software layer between the application and the hardware and use machine resources, and in particular memory. For example, DynamoRIO has been shown to have a 500% memory overhead [9]. To address this issue, we explored memory optimization opportunities presented by exit stubs in code caches.

A code cache is used in most VEEs to store both application code and exit stubs. If the next instruction to be executed is not in the code cache, exit stubs (or trampolines) are used to return control to the SDT to fetch the next instruction stream. It is beneficial to reduce the space demands of a code cache. First, a small code cache reduces the pressure on the memory subsystem. Second, it improves instruction cache locality because the code is confined to a smaller area within memory, and is therefore more likely to fit within a hardware cache. Third, it reduces the number of cache allocations and evictions. Solutions for managing the size of code caches (using eviction techniques) have been proposed elsewhere [8, 7], yet those studies focused on code traces.

Exit stubs typically have a fairly standard functionality. They are duplicated many times in the code cache and are often not used after the target code region is inserted into the code cache, providing ample opportunity for optimization. We explored the memory overhead of stubs in a SDT and evaluated techniques for minimizing that space. We developed three techniques that work for both single-threaded and multi-threaded programs. The first technique deletes stubs that are no longer needed but assume unlimited code caches and the absence of flushing. It removes stubs in their entirety when these stubs are guaranteed not to be needed anymore. Although there are many such applications which do not violate these assumptions and still have a reasonable code cache size, it is also important to be able to handle situations where flushing occurs. The last two techniques lift these restrictions and identify stub characteristics that can reduce space requirements.

Our experiments showed these techniques can reduce memory consumption by the code cache by up to 43% and, in some cases, yield performance improvements as well. Our experiments also showed that performance improvement is even better for limited size code caches which are used when the constraints on memory are even more severe.

3. An Instrumentation Tool for Virtual Execution Environments

With dynamic translation, a program in a SDT has two binaries: an input source binary and a dynamically generated target binary. Program analysis is important for these binaries. However, existing instrumentation systems for use in SDTs have two drawbacks. First, they are tightly bound with a specific SDT and thus are difficult to reuse without extensive effort. Second, most of them can not support instrumentation on both the source and target binaries.

In this project, we present Dimension, a tool that provides instrumentation services for SDTs. The objective for Dimension is to build a flexible and efficient instrumentation tool that can be used by a variety of SDTs. Given an instrumentation specification, Dimension can be used by a SDT to provide customized instrumentation, enabling analyses on both the source and target binaries. The design of Dimension identifies the few components of SDTs that need to communicate with an instrumenter and develops interfaces between the SDT and Dimension.

To avoid interfering with a SDT's code generation and code cache management mechanisms, Dimension uses the probe-based instrumentation technique, replacing a program's binary instructions with jumps to invoke the instrumentation code [13]. Therefore, a SDT needs to provide the location of both the source and target code to Dimension, and Dimension then can analyze the instructions and make instrumentation decisions according to an instrumentation specification.

When a user wants source binary instrumentation, one approach actually instruments source binary before translation and the instrumented code is then translated and executed. Extra translation overhead is paid for instrumentation code in this approach. Another scheme modifies the target binary after translation, since the effects of source binary instrumentation can always be achieved by instrumenting the corresponding target binary. This method requires the mapping from the source instructions to the target instructions, and a SDT needs to provide the source-to-target mapping to the instrumentation system. Compared to the first approach, the second method avoids the extra translation overhead, but has more communication overhead. Dimension uses the second approach for simplicity.

We performed experiments to determine the efficiency of Dimension. Three different experiments were performed to evaluate Dimension's performance from different reference points. The first experiment focuses on the instrumentation optimization mechanisms and included inlining and partial context switching. Each of the optimizations is evaluated to determine its effectiveness to improve Dimension's performance. In the second experiment, the same instrumentation that was manually implemented in Jazz is performed automatically by interfacing with Dimension. By comparing the two performances, Dimension is evaluated to see how its portability feature can affect its performance. In the last experiment, we demonstrate that the use domain of Dimension can be extended to provide efficient instrumentation. Without any optimization, the slowdown is fairly large, up to 16.2x in Strata (gzip) and 4.5x in Jikes RVM (mpegaudio). Inlining helps, with the average slowdown improving from 8.6x to 6.4x in Strata and from 2.4x to 2.1x in Jikes RVM. A significant performance improvement comes from partial context switch, which reduces the average slowdown to 2.6x in Strata and 1.4x in Jikes RVM. Probe coalescing finally reduces it to 2.0x in Strata and 1.1x in Jikes RVM.

The optimizations applied in Dimension effectively reduce the slowdown from instrumentation in both the source and target binaries. Their benefits are consistent across the two different SDTs but, as usual, depend on the number of optimization opportunities.

In the second experiment, we investigate how Dimension's flexibility affects the efficiency, i.e., the performance difference between Dimension and instrumentation systems that are built as instrumentors. We use Dimension to perform the same instrumentation that has been manually implemented on a SDT and compare their performances. Jazz [18] builds a branch coverage tester by instrumenting each basic block of a Java program to determining what edges are covered by an execution. We use Dimension to build the same instrumentation for Jikes RVM and compare their performances. To our knowledge, Jazz uses all the instrumentation optimization techniques that Dimension uses, so the comparison can fairly illustrate the trade-off between efficiency and flexibility. Our results show that Dimension achieves comparable performance against the systems in which instrumentation is manually built into a VEE. The flexibility of Dimension introduces negligible effect on its performance.

In the third experiment, we used a standard, basic-block counting instrumentation, to compare the performance of Strata-Dimension against Pin, Valgrind, and DynamoRIO. The metric used is the slowdown from instrumentation, which is the execution time with instrumentation normalized to the native execution. The data for Valgrind, DynamoRIO and Pin were obtained from a public publication [16]. As Figure 2 illustrates, Strata-Dimension introduces a reasonable slowdown from instrumentation. The average slowdown for Strata-Dimension is 2.6x, which is slightly worse than Pin (2.3x) but better than both DynamoRIO (4.9x) and Valgrind (7.5x). Pin performs more complicated optimizations on instrumentation than Dimension does. DynamoRIO is primarily designed for dynamic optimization, and does not automatically perform instrumentation optimizations (e.g., partial context-switch) like Dimension and Pin. We believe the slowdown of Valgrind mainly comes from its overall infrastructure, which has a slowdown of 5.6x even when program is executed without instrumentation. The experiment demonstrates that Dimension can be used to provide efficient dynamic instrumentation tools in traditional execution environments.

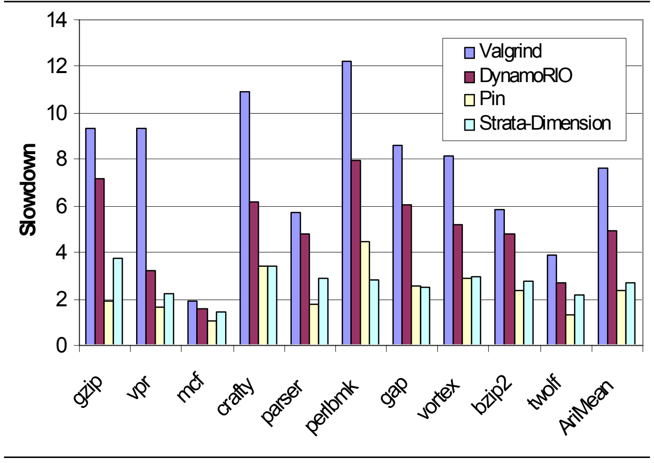

Figure 2: Comparison of slowdown from instrumentation in traditional execution environments.

As the experimental results show, Dimension achieves flexibility and efficiency simultaneously. The optimization techniques applied by Dimension effectively reduce the slowdown from instrumentation, leading to a comparable performance against systems in which instrumentation is developed for a particular VEE. The use domain of Dimension can be easily extended, e.g., building efficient regular instrumentation systems for use in traditional execution environment.

4. Source-level Debugging for Trace-based Dynamic Optimization

We also developed a dynamic optimizer and a debugger for the dynamically optimized code. Debugging programs at the source level is essential in software development, as the user is familiar with the source level code but not the executing code. With the growing importance of dynamic optimization, there is a clear need for debugging support in the presence of runtime code transformation. This work presents the first debugging framework called DeDoc, that allows debugging at the source level for programs that have been transformed by a trace-based dynamic binary optimizer. Our techniques provide full transparency and hide the effect of dynamic optimizations on code statements and data values from the user. We describe and evaluate an implementation of DeDoc. Our experiments show that nearly all values can be reported from a dynamically optimized program with DeDoc. The results also show that our techniques are practical, with a minimal performance overhead of up to just 2.6% when generating debug information during program execution.

The key challenge in developing a debugger for dynamic optimization is how to provide for online communication between the dynamic optimizer and the debugger. An active debug environment is needed in which the debug information reflects generation, modification and deletion of code by a dynamic optimizer and is immediately available to the debugger. The second challenge is the code location and data value problems in the presence of re-optimization and trace combination. Another challenge is that debug information must be generated with low overhead. This challenge emerges due to the frequent and fine-grain application of optimizations on traces. There is also much more code duplication as many traces may cover the same basic blocks. Further, traces may be periodically flushed (deleted).

Our DeDoc framework provides dynamic debug mappings that track code modifications and code duplication as optimizations are applied to the executing code. These mappings can be generated, modified and communicated to the debugger efficiently. DeDoc uses several techniques to generate additional lightweight debug information with "annotations" that are attached to executing code through invisible breakpoints. These annotations extract expected program values to handle the data value problem. The debug information consisting of the mappings and annotations is also needed to handle re-optimizations and code granularity changes. Lastly, DeDoc is retargetable to support existing dynamic optimizers and existing debuggers.

DeDoc addresses code location and data value problems posed by trace-based dynamic binary optimizers. Code location problems are handled by debug mappings, and data value problems are handled by placing annotations in the code to extract variables values at runtime. The mappings and annotations can be generated and modified during program execution. A component of DeDoc, the debug engine, computes the mappings and annotations using information provided by the dynamic optimizer. The debug engine facilitates the communication of runtime debug information to the debugger.

A salient feature of DeDoc is that it intercepts actions performed by an existing debugger on a program and hides the effects of dynamic optimization from the debugger. We developed an implementation of DeDoc using a dynamic optimizer and the Gdb debugger. We also showed that our techniques can report most program values and have very low overhead.

5. Related work

Software dynamic translation has been used for a number of purposes (see Section 1), including dynamic binary translation of one machine instruction set to another [22], emulation of operating systems (e.g., VMWare, Plex86) [19], and machine simulation [3, 23]. While most of these systems have been built for a single purpose, there has been recent work on general infrastructures for SDT that are similar to Strata.

Walkabout is a retargetable binary translation framework that uses a machine dependent intermediate representation to translate and execute binary code from a source machine on a host machine [22]. It analyzes the code of the source machine to determine how to translate it to the host machine or to emulate it on the host.

Another flexible framework for SDT is DynamoRIO [2], which is a library and set of API calls for building SDTs on the x86. One such system built with DynamoRIO addresses code security. Unlike Strata, to the best of our knowledge, DynamoRIO was not designed with retargetability in mind. Another difference is that DynamoRIO is distributed as a set of binary libraries. The source code is available for Strata, making it possible to modify and experiment with the underlying infrastructure to implement new SDT systems.

Pin is a VEE that provides transparent and portable instrumentation [16]. It was built for four different architectures to demonstrate how little effort is made to port to different platforms, including IA-32, EM64T, Itanium, and ARM architectures. Pin uses dynamic compilation to instrument executables as they are executing.

To achieve high performance in a SDT system, it is important to reduce the overhead of the translation step. For a retargetable and flexible system like Strata, it can be all the more difficult to achieve good performance across a variety of architectures and operating systems. A number of SDT systems have tackled the overhead problem. For example, Shade [3] and the Embra [23] emulator use a technique called chaining to link together cache-resident code fragments to bypass translation lookups. This technique is similar to one of the overhead reduction techniques in Strata that links a series of fragments to avoid context switches.

The most closed related work in debugging optimized code was done by Jaramillo et al. in the Fulldoc debugger [12]. Jaramillo described mappings that can relate every instance of a statement in optimized code with an unoptimized counterpart. Fulldoc proposed *late* and *stop* annotations to extract data values whose reportability is affected due to optimizations. DeDoc builds on Fulldoc's techniques, particularly its data tracking scheme.

Some VEEs have been manually extended to provide instrumentation. Jazz [18] is a program testing tool which builds instrumentation functionality in Jikes RVM for structural testing. DynamoRIO was extended from a dynamic optimization system, with the addition of a set of instrumentation APIs for building customized program analysis tools. Both Jazz and DynamoRIO provides customized instrumentation by extending the original translation-based VEE. However, designs and implementations are for one particular system, which lacks the portability to a wide variety of different translation-based VEEs.

6. Conclusions and Future Work

With this work, we have demonstrated that SDTs can be made efficient. We also showed that tools that support the use of SDTs are viable.

There are a number of challenges that we believe can be tackled with SDTs. In our next step, we will focus on one of them, process variation. With the emergence of chip multiprocessors (CMPs), comes the promise of high-performance computing on a desktop. However, an inherent characteristic of CMPs that presents a significant obstacle is **process variation:** Timing and power consumption will vary across identically designed components of a CMP, producing a negative impact on application performance and ultimately limiting the benefit of CMPs. Process variation has been identified as one of the key problems that could block further scaling of circuits if not addressed.

In our next step of research, we plan to explore an approach that combines the hardware, compiler and OS. Namely, we propose to use an advanced execution system, called a *Robust Execution Environment (REEact)*, that will dynamically adapt an application's execution to the runtime resource landscape originating from process variations. REEact will be a type of virtual execution environment (VEE) that mediates, controls and adapts the application's execution. It will employ a combination of techniques in adapting both the hardware resources and the application software code to overcome the impact of process variations. At the hardware level, it will adapt the resources, such as setting the speed/voltage of a node on the CMP. At the software level, REEact will dynamically optimize code, taking into account performance and power consumption due to process variations and environmental effects. REEact will even elicit the help of the OS in determining what resources to use in running the application. It will also inform the OS about information it dynamically discovers about latency, power, and application behavior.

References

[1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 1–12, New York, NY, USA, 2000. ACM Press.

[2] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[3] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128–137, New York, NY, USA, 1994. ACM Press.

[4] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society.

[5] James Gosling. *The Java Language Specification*. Addison-Wesley, Reading, MA, USA, 2000.

[6] Kim Hazelwood and David Brooks. Eliminating voltage emergencies via microarchitectural voltage control feedback and dynamic optimization. In *ISLPED '04: Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, pages 326–331, New York, NY, USA, 2004. ACM Press.

[7] Kim Hazelwood and James E. Smith. Exploring code cache eviction granularities in dynamic optimization systems. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, page 89, Washington, DC, USA, 2004. IEEE Computer Society.

[8] Kim Hazelwood and Michael D. Smith. Generational cache management of code traces in dynamic optimization systems. In *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 169, Washington, DC, USA, 2003. IEEE Computer Society.

[9] Kim Hazelwood and Michael D. Smith. Managing bounded code caches in dynamic binary optimization systems. *ACM Transactions on Architecture and Code Optimization*, 3(3):263–294, 2006.

[10] Jason D. Hiser, Daniel Williams, Adrian Filipi, Jack W. Davidson, and Bruce R. Childers. Evaluating fragment construction policies for sdt systems. In *VEE '06: Proceedings of the 2nd International Cnference on Virtual Execution Environments*, pages 122–132, New York, NY, USA, 2006. ACM Press.

[11] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *VEE '06: Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 2–12, New York, NY, USA, 2006. ACM Press.

[12] C. Jaramillo, R. Gupta, and M. L. Soffa. FULLDOC: A full reporting debugger for optimized code. In *Proceedings of the Static Analysis Symposium*, volume 1824, pages 240–259. Springer, 2000.

[13] Peter B. Kessler. Fast breakpoints: Design and implementation. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 78–84, New York, NY, USA, 1990. ACM Press.

[14] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. USENIX '02: Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[15] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86, Washington, DC, USA, 2004. IEEE Computer Society.

[16] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[17] J. Maebe, M. Ronsse, and K. De Bosschere. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Compendium of Workshops and Tutorials held in conjunction with PACT 2002*, 2002.

[18] Jonathan Misurda, James A. Clause, Juliya L. Reed, Bruce R. Childers, and Mary Lou Soffa. Demand-driven structural testing with dynamic instrumentation. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 156–165, 2005.

[19] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation or Building Tools is Easy.* PhD Dissertation, University of Cambridge, Cambridge, UK, November 2004.

[20] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society.

[21] Kevin Scott and Jack W. Davidson. Safe virtual execution using software dynamic translation. In *Proceedings of the 18th Annual Computer Security Applications Conference*, pages 209–218, Las Vegas, NV, December 2002.

[22] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. In *DYNAMO '00: Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 41–51, New York, NY, USA, 2000. ACM Press.

[23] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 68–79, New York, NY, USA, 1996. ACM Press.