Instrumentation in Software Dynamic Translators for Self-Managed Systems

Naveen Kumar, Jonathan Misurda, Bruce R. Childers
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
{naveen,jmisurda,childers}@cs.pitt.edu

Mary Lou Soffa
Department of Computer Science
University of Virginia
Charlottesville, Virginia 22904
soffa@cs.virginia.edu

Abstract

Self-managed software requires monitoring and code changes to an executing program. One technology that enables such self management is software dynamic translation (SDT), which allows a program's execution to be intercepted and controlled by a separate software layer. SDT has been used to build many useful applications, including software security checkers that check for code vulnerabilities, dynamic code optimizers, and program introspection tools. While these systems use program instrumentation, the instrumentation is usually tailored to a specific application and infrastructure. What is missing is a single scalable and flexible instrumentation framework that can be used in different self-managed SDT infrastructures. In this paper, we describe such a framework, called "FIST," that can be used and targeted by different algorithms and tools to enable instrumentation applications that are portable across SDTs and machine platforms. Our interface supports multiple levels of granularity from source level constructs to the instruction and machine level. We describe and evaluate FIST's capabilities in the Strata system for the SPARC and the Jikes Research Virtual Machine for Java on the Intel x86.

1. Introduction

Self-managed software systems that dynamically modify and control the execution of a program have received much attention due to the increased recognition of their importance. For example, dynamic optimizers, such as IBM's Jikes optimizer [1] are self-managing systems that decide when to apply code transformations at run-time based on program behavior. In Jikes, decisions about how to optimize a method are based on the predicted benefit of the optimization and its cost. Other examples of such *self-managed software dynamic translators* (SDT) have been used for detection and repair of program faults [5,7], enforcing security policies [11], dynamic compilation and optimization of binary and Java programs [1], and debugging programs [9]. All of these systems use information about the executing program to make *decisions* about how to *control* the program's execution and what to change. For example, fault detectors check invariants by instrumenting a program to monitor for anomalous behavior [6], program debuggers insert instrumentation to track values [9] (possibly even identifying sources of errors [18] or applying automatic repairs [5]), and security checkers use instrumentation to check for and recover from vulnerabilities [11]. Instrumentation in these and other systems is used to both gather information and control or modify—i.e., self manage—the executing program.

Many instrumentation techniques and systems have been proposed to monitor and control a program's execution. These techniques include static binary rewriting of application code for profiling [12,17] and dynamic instrumentation [8,13,14,15]. There have also been infrastructures that aim to provide instrumentation capabilities for different machine platforms [8,12]. However, these system lack a common interface and general framework for dynamic instrumentation in SDT. Yet, with the importance and number of self-managed applications that use instrumentation and SDT, there is a need to provide a framework that can be used for different instrumentation and control purposes. Such a framework must be *flexible* to be configured for different purposes, architectures, and SDTs and be *scalable* for different granularities and amounts of information gathering and controlling.

This paper describes **FIST**: a new **F**ramework for program **I**nstrumentation in self-managed **S**oftware dynamic **T**ranslators that is both scalable and flexible. The framework can gather information, control a program, and dynamically adapt the code and instrumentation. FIST is portable across different infrastructures and machine architectures: It has a consistent and single interface for instrumentation that avoids tying instrumentation algorithms and tools to a single SDT infrastructure or machine architecture. For example, a security checker that uses our instrumentation capabilities can be hosted in any SDT.

This paper makes several contributions, including:

- A framework (FIST) that combines an event-response model and instrumentation primitives to enable flexible and scalable information collection and control in self-managed systems that use SDT,
- Instrumentation primitives that are portable across different SDT infrastructures and machine architectures with variable length and fixed-length instruction sets,
- An instance of FIST for a software dynamic translator (Strata) and the SPARC,
- A second instance of the framework for a JIT for Java (Jikes RVM) for the Intel x86, and
- An evaluation of the performance and memory overhead of our primitives.

Such a flexible and scalable instrumentation framework can be configured for many applications and uses. For example, it can be used to enforce security policies by monitoring program vulnerabilities [11]. In these security applications, monitoring is done for program vulnerabilities at the instruction level, and when a potential vulnerability is found, more aggressive sandboxing can be applied. FIST can also be used to modify the behavior of a program to correct an error. Indeed, there are many compelling SDT applications that could benefit from a general framework, ranging from code profilers, to software testing tools, and to adaptive code environments.

WOSS'04 Oct 31-Nov 1, 2004 Newport Beach, CA, USA

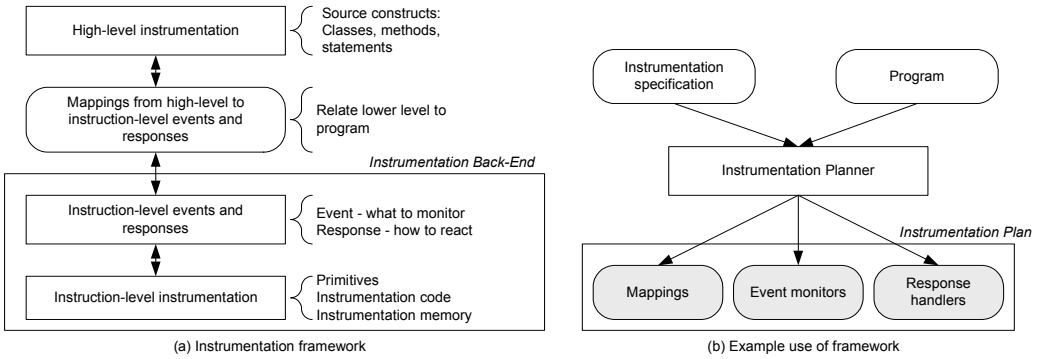

Figure 1: FIST and an example of its use

The remainder of this paper has the following organization. Section 2 describes FIST, while Section 3 describes the challenges and issues associated with incorporating the framework into a SDT system and a Java VM. Section 4 describes related work and Section 5 summarizes the paper.

2. FIST

Our instrumentation mechanism is lightweight and allows trade-offs between the cost and amount of interaction with the program. It also supports algorithms and techniques that monitor, profile, and affect program execution at different levels in different ways. The mechanism can dynamically insert and remove instrumentation to make decisions about how to instrument and control program execution based on run-time behavior. It also avoids exposing machine or platform characteristics. To achieve these capabilities, the framework uses an *event-response model* that triggers information gathering and control when a property about a running program is satisfied. In our approach, an *event* occurs when a program *monitor* discovers that a property of the running program is satisfied and a *response* is taken for that event. This reactive model permits controlling the instrumentation dynamically and affecting the program execution in different ways. Instrumentation for security checks verify that system calls meet a security policy, and if a call is unsafe, a response is taken, such as aborting the program with an error message. Here, an event is triggered on a policy violation and a response handles the event, which in this case aborts the program.

Figure 1(a) shows the components of FIST. The framework permits information gathering and control at the high level for constructs such as classes, methods, and statements. Programs execute at the instruction level and source information must be related to the instructions and a program's execution. FIST has *mappings* that indicate how high-level constructs are related and translated to the instruction level. To instrument an application, the framework has primitives to monitor and gather information and to check whether specified properties are satisfied. The primitives integrate monitoring, events, and responses at the instruction level and they are the mechanism by which an instrumentation application is implemented. The combination of mappings and instrumentation code built with our primitives form an *instrumentation plan* that says how to gather information and control a program's execution. The instrumentation plan is a "recipe" of how to instrument a program and the primitives are the "steps" that say exactly what to do.

With FIST, standalone tools can target our interfaces to gather information and control a program. Figure 1(b) shows a possible way in which the framework can be used. In the figure, an instrumentation planner translates an instrumentation specification into an instrumentation plan. The plan includes mappings to relate instruction-level information to the source level. There are also event monitors and response handlers which are pieces of instrumentation code that can be stitched into the program's execution with our instrumentation primitives. In a later section, we present an application of our framework to software structural testing of Java programs that uses an approach similar to Figure 1(b).

We focus on the back-end portion of FIST in this paper because it is the basis upon which the rest of the framework is built. We describe the components of the back-end, including its mechanism for generating events and responding to them, the primitives for integrating events and responses, and instrumentation memory for code and data is handled.

2.1. Event Generation and Response

In FIST, events are generated synchronously during a program's execution at well-defined points in the code. A condition for generating an event is a boolean expression that can be evaluated in the context of the running program and machine platform. A response is general and may do any number of activities for an event, such as changing the instrumentation sampling rate, saving state about the program and machine platform, or updating the program with a repair.

Events have *static* and *dynamic* properties that need to be monitored. A static property can be verified without actual values or state at a particular instrumentation point. For example, instruction type is a property that can be determined strictly by looking at instructions without knowing anything about their dynamic execution. A dynamic property can only be verified by inspecting program values and state at an instrumentation point. For example, the data value of a particular machine register or memory location is a dynamic property. Instrumentation that monitors properties can be inserted at compile-time or dynamically inserted and removed at runtime. Further, the instrumentation itself can inspect instructions and state, and insert or remove instrumentation.

To implement events and responses, FIST has a *trigger-action* mechanism. A *trigger* is code that checks for some specific condition and generates an event, while an *action* is code that reacts to events. When a property is satisfied, the trigger generates an event, which causes a call-back to the action. The code that does event generation and call-back to actions is a "trigger-action pair".

The trigger-action mechanism works by attaching an instrumentation point to the binary program. This instrumentation point, when executed, transfers control to a trigger, which checks for a code property and generates an event when that property holds. When a trigger is fired, an action is taken for the event. The figure shows that triggers can be shared by instrumentation points and actions can be shared by different triggers. An instrumentation point can also invoke several triggers and their corresponding actions.

To monitor properties, a trigger has a *static check* and a *dynamic check*. The static check verifies static properties about instructions, the machine and the dynamic translator. It is done when inserting new instrumentation into the program.

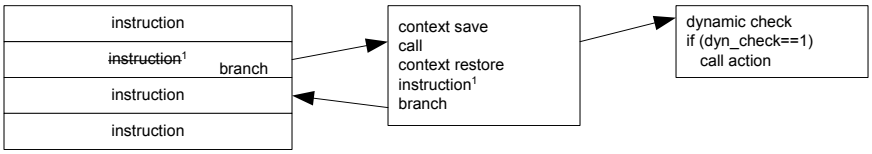

Figure 2: Hit-many for a dynamic check on a fixed-length instruction set

A trigger's dynamic check inspects values and state at run time and is invoked when control flow reaches an instrumentation point in the program. Static checks are implemented as part of the instrumentation system and interface, and dynamic checks are implemented with *instrumentation primitives*. These are the mechanisms that integrate monitoring, event generation, and responses to intercept program control flow to execute dynamic checks and actions.

2.2. Instrumentation Primitives

FIST has three primitives for instrumentation: *inline-hit-always*, *hit-once*, and *hit-many*. These primitives are used to build more complex operations and they differ in the way in which instrumentation is inserted and left in the application. *Inline-hit-always* is inserted directly into a basic block and never removed. *Hit-once* is executed outside of the program control flow and it is removed immediately after being hit. Similarly, *hit-many* is executed outside of regular control flow and remains in the code until explicitly removed.

Hit-once and hit-many intercept control flow and change it to go out-of-line to another location. These primitives use a fast breakpoint that replaces an instruction by a branch [10] that takes control to code that does the dynamic check.

Figure 2 shows how we use fast breakpoints for a dynamic check. As the figure shows, the breakpoint code (a breakpoint handler) saves the context of the application, makes a call to a function to do the dynamic check, restores the context of the application, executes the original instruction and then executes the next instruction after the one that was instrumented. The application context here refers to the general-purpose registers and other machine registers like the condition code. This context must be saved before invoking the dynamic check or action to free registers for the dynamic check and action code. The context is available to be inspected by a dynamic check and modified by the action, if desired. For inline-hit-always, the code to save and restore the context is inserted in-line during code generation around the call to the dynamic check. Inline instrumentation eliminates at least two branch instructions. Inline-hit-always also has the ability to include simple instrumentation directly in the code, rather than transferring control to the dynamic check and action.

The advantage of *hit-once* and *hit-many* is they can be inserted and removed dynamically. Using a fast breakpoint to implement these primitives makes it easy to insert a primitive without changing code layout. To insert *hit-once* or *hit-many*, usually only one instruction has to be changed. Likewise, removing instrumentation is easy because the original instrumented instruction can be copied back to the instrumentation point to remove the instrumentation. The only primitive that is ever inserted in-line is one that will always remain in the code. This avoids dynamically rewriting the code because inline instrumentation is never removed once inserted. With these primitives, we can implement other primitives. For example, *hit-many* can be used to implement a *hit-always* primitive that is dynamically inserted and never removed.

2.3. Instrumentation Code and Data Memory

Instrumentation should not disturb a program to avoid introducing artificial effects. However, in practice, it is difficult to completely avoid disturbing the program. We minimize the disturbance by using a separate context for instrumentation code and data values. To keep the instrumentation lightweight, this context is kept in an application's process space, which avoids expensive process switches and inter-process communication between the application and the instrumentation. Context management is done by the instrumentation itself, and depending on the particular target platform, we keep the memory for instrumentation code in a single memory region or attached to individual functions or methods. Instrumentation typically gathers information and needs to store that information some place. Our framework provides a separate data memory region that holds persistent information and variables needed by the instrumentation.

3. SDT Instrumentation

To demonstrate FIST's flexibility, we have incorporated it in a SDT for the SPARC and a Java JIT/VM for the x86. In this section, we describe the challenges associated with each implementation, including special considerations for the target instruction set architectures (ISA) and SDT.

3.1. Strata and SPARC/Solaris

To address the difficulty of building software dynamic translators, we (with the University of Virginia [16]) developed a highly configurable and retargetable SDT infrastructure called Strata. Strata is arranged as a VM that sits between the program and the CPU. The VM translates a program's instructions before they execute on the CPU and mimics the standard hardware with fetch, decode, translate and execute steps. Fetch loads instructions from memory, decode cracks instructions into their individual fields, and translate does any modifications to the instructions as they are written into a *fragment* or *trace cache*. The translate step is the point at which the code can be modified. The execute step occurs when control is returned to the binary in the fragment cache. To include our instrumentation framework in Strata, we had to address how to incorporate it into the VM and how to store instrumentation code and data as part of the VM.

VM Interface. The Strata VM has target-independent common services, target-dependent specific services, and an interface through which the two communicate. We incorporated our instrumentation mechanisms in the common and target specific services. The common services provide the interface to convey to Strata the static and dynamic checks and actions. The interface passes function pointers for call backs to check a static or dynamic property and to invoke an action. Hence, the static and dynamic checks and the action can be functions written in a high-level language. The interface also allows dynamic insertion and removal of instrumentation and exports all program and machine state, and Strata internal structures. The target specific services define an interface for inserting and removing *hit-once* and *hit-many* primitives on a specific machine to ease retargetability of the infrastructure.

Instrumentation Code and Data Memory. Code storage for breakpoints is allocated in Strata's fragment cache. This space may be located in a portion of the cache reserved for instrumentation or immediately after a fragment in the cache. The former has the advantage that code layout is preserved for instruction traces, while the latter has locality benefits when instrumentation code is executed frequently with a fragment. When adding new dynamic checks on-the-fly into already existing fragments, space is always allocated in the reserved

portion of the fragment cache. Instrumentation data memory is also allocated as part of Strata's context. An interface exports access to this memory to the instrumentation.

Experiments. We measured the average memory and performance overhead of the instrumentation primitives for Strata on the SPARC, as shown in Table 1. The memory cost is the number of instructions needed by each primitive, while the performance cost is the run-time to execute a primitive. The results were collected on a 500 MHz Sun Blade 100 with 256 MB of RAM. To compute overhead, we used a loop that iterated 100 million times. On every iteration, a primitive is inserted, removed and executed. Only the costs associated with the primitive is measured.

	Hit-once	Hit-many	Inline-hit-always
Time (ns)	660	640	510
Num. instrs.	72	53	49

Table 1: Overhead of instrumentation primitives

Most of the instrumentation expense is due to saving and restoring context. A full context save or restore takes 21 instructions each and the call to the dynamic check and action takes 7 instructions. The control transfers to and from the breakpoint handler take another 4 instructions and the cost of emitting code at run-time for *hit-once* is 14 instructions for the first instruction and 5 for each additional instruction.

In all cases, the performance cost of the instrumentation is compounded by the presence of control transfers. *Hit-once* also does a cache flush when the original instruction is replaced. *Inline-hit-always* is the least expensive because it has two less branches and DSIs than *hit-once* and *hit-many.* Nevertheless, *hit-many* has good performance because it can be both dynamically inserted and removed on-the-fly and *hit-once* has low cost for temporary instrumentation because it removes itself without any further cost.

To investigate the probe overhead in full applications, the SPEC2000 benchmarks were instrumented with *hit-many* probes. To compute the average probe cost, we inserted a *hit-many* probe into each basic block executed. The dynamic check always invoked the payload and the payload did no operation. To compute the overhead, the difference in run-times of the instrumented and the non-instrumented code was divided by the total number of probes executed.

Benchmarks	**Time (ns)**	**Benchmarks**	**Time (ns)**
mcf	104	*vortex*	90
bzip	82	*perl*	93
gzip	87	*gcc*	106
vpr	153	*twolf*	103
		Average	102

Table 2: Hit-many instrumentation probe cost

Table 2 shows the overhead the benchmarks. The second column is the cost of a probe that does a full context save and restore. A full context switch on the SPARC saves and restores all machine registers. As the table shows, the probe cost varies from 81–152 ns (average 102 ns) for a full context switch. The large variation in the probe cost depends on how well the cost of instrumentation insertion is amortized by frequent execution of the probe. The overhead for *hit-once* and *hit-always* is similar to *hit-many* because the cost of inserting the *hit-many* probes is quickly amortized. The cost of a probe also depends on machine affects, such cache misses and branch mispredictions.

3.2. Jikes RVM and x86/Linux

FIST for the Jikes Research Virtual Machine (RVM) [1] instruments an executing Java program at the instruction level on x86, and can insert and remove instrumentation at any point during a program's execution. We can also map instruction-level information to bytecode and source statements, provided the bytecode has line number mappings. To integrate FIST into the Jikes RVM, we had to address three issues. The first one was how the instrumentation system gets control to instrument a method, the second was how to handle multi-threading, and the final issue was the interaction of garbage collection (GC) and instrumentation.

Instrumentation Injection. To get control when a method is compiled, a simple modification was made to the VM to insert a breakpoint in a method's prologue to generate an event when a method is entered. The response for that event gets control on method entry. This structure makes the instrumentation independent and transparent to the VM: the only interaction is the initial insertion of the static breakpoint. This approach also ensures that methods are only instrumented when they are executed. Finally, the JIT exports information to the instrumentation, such as a method's control flow graph.

Multi-threading Support. FIST supports multi-threading as found in Java programs. Multi-threading comes into play when trying to track state in a method with instrumentation code. Because we use source-sink breakpoints, it is possible that a thread switch can happen between the execution of the source and sink breakpoint. In such a case, when the source breakpoint needs to pass state to the sink breakpoint, the state must be saved as part of the thread context. One possibility is to modify the thread switch code in the Jikes RVM to save this state. However, the state is likely to be instrumentation application dependent and it is impractical to modify the RVM whenever a new instrumentation application is developed. Instead, yield points can be automatically identified and special context saving instrumentation inserted to record state that needs to be persistent across thread switches.

Garbage Collection. The concern with GC is where to allocate data and code space for the instrumentation. One possibility is to allocate storage as part of the application context or in the context of the Jikes RVM. This solution may, however, have interactions with GC. A problem is that the instrumentation is machine code and GC may not be able to track references involving the instrumentation. Another problem is that in copying GC, addresses can change. However, for efficiency, it is desirable to emit address constants in the instrumentation, rather than doing a table lookup to find addresses. Because the solutions to these problems were expensive, we rejected allocating instrumentation data and code as part of the application. Instead, we allocate a memory buffer from the operating system that is not visible to the run-time system to hold instrumentation code and data. It avoids any interactions with GC.

Experiments. For the x86, we measured the ideal performance of the instrumentation primitives with a tight loop whose body was instrumented (the same experiment as described earlier for Strata). Our experimental platform was Jikes 2.1.1 and a 700 MHz Pentium III machine with 512 MB memory and RedHat Linux 2.4.18.

	Hit-once	Hit-many	Inline-hit-always
Time (ns)	469	939	65
Num. instrs.	25	25	21
Storage	11	11	0

Table 3: Overhead of instrumentation primitives

Table 3 shows the average cost of the instrumentation primitives. The cost of *hit-many* is twice the cost of *hit-once* because two breakpoints (a "source" and a "sink" breakpoint) are executed by *hit-many* to ensure that the primitive remains until explicitly removed. The use of source-sink breakpoints is a consequence of the x86 instruction set architecture. *Hit-once* uses the same mechanism as *hit-many*, except it does not

use a "sink" breakpoint. *Inline-hit-always* is faster than both *hit-many* and *hit-once* because it does not make code modifications. *Hit-many* and *hit-once* use self-modifying code, which incurs significant cost because the instruction and data caches must be kept consistent by the hardware. The table also shows memory overhead. *Hit-once* and *hit-many* both use 25 instructions (68 bytes). There is an average of 11 bytes of data storage to hold the original instruction and the addresses where sink breakpoints should be inserted. *Inline-hit-always* does not have any storage cost because the primitive is inserted inline in the code and does not insert successors.

We also measured the average overhead of the *hit-many* in several programs from SPECjvm98, as shown in Table 4. These results were obtained from a tool that we developed to do branch coverage testing. The numbers reflect expected performance of *hit-many*, including all machine effects. As the table shows, the instrumentation overhead can vary considerably, ranging from 937 to 4,117 ns (average is 1,849 ns). The large variance is partly due to instruction cache and branch prediction behavior. In programs with tight loops (e.g., *compress* and *mpegaudio*), *hit-many* has better cache and branch behavior, which results in less overhead.

Benchmarks	Time (ns)	Benchmarks	Time (ns)
jess	2,890	*db*	1,030
javac	1,637	*mpegaudio*	1,114
mtrt	1,221	*jack*	4,117
compress	937	**Average**	1,849

Table 4: Hit-many instrumentation probe cost

A second effect is related to the breakpoints used by hit-many. This primitive requires a sink breakpoint at each successor basic block to an instrumentation location. Hence, the more successors, the larger the cost of *hit-many* and performance tracks the average number of successors. For *jack* and *jess*, the number of successors is quite high (both have many control transfers), and *hit-many* has a larger cost because many sink breakpoints are inserted. In *compress* and *mpegaudio* there are fewer sink breakpoints, and combined with good machine behavior, *hit-many* has better performance.

4. Related Work

Instrumentation techniques have been used in SDTs for a number of purposes, including dynamic optimization [1,2], program debuggers [9], software security [11], and binary translation [4]. In these systems, the instrumentation is hard coded. In Dynamo [2], the instrumentation happens in the interpreter and in Dynamo/RIO, instrumentation is inserted on back edges inside a basic block [3].

Fast breakpoints were pioneered by Kessler [10] and used the technique that we call *hit-many* instrumentation. Kessler's fast breakpoints were not applied across different SDT and architectures to dynamically instrument programs. Systems like Dyninst [8] and Paradyn [13] use fast breakpoints similarly to our approach. Like our framework, Dyninst is general, with a language for specifying instrumentation [8]. However, an approach for different SDTs with Dyninst has not been described. Other instrumentation toolkits like ProbeMeister [15] and Pin [14] are targeted to specific frameworks (e.g., the Java VM for ProbeMeister) and are not portable across SDT infrastructures.

5. Summary

This paper described a framework for flexible and scalable instrumentation, called FIST, in self-managed software dynamic translators. We demonstrated FIST's capabilities in two different SDTs and two target architectures. We described how our framework and primitives can be incorporated in the Strata SDT for the SPARC and in a Java VM for the x86. These framework instances show that our approach is indeed flexible enough to be used by different SDTs.

6. Acknowledgements

This work was supported in part by National Science Foundation grants ACI–0305198 and ACI–0203945 and an IBM Eclipse Innovation Grant.

7. References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, P. Sweeney, "Adaptive optimization in the Jalapeño JVM", *Conf. on Object Oriented Programming, Systems, Lang. and Applications,* 2000.

[2] V. Bala, E. Duesterwald,, S. Banerjia, "Dynamo: A transparent dynamic optimization system", *Conf. on Programming Lang. Design and Implementation*, 2000.

[3] D. Bruening, T. Garnett, S. Amarasinghe, "An infrastructure for adaptive dynamic optimization", *Int'l. Symp. on Code Generation and Optimization*, 2003.

[4] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, et al., "The Transmeta Code Morphing Software: Using speculation, recovery, and adaptive retranslation to address real-life challenges", *Int'l. Symp. on Code Optimization and Generation*, 2003.

[5] B. Demsky, M. Rinard, "Automatic detection and repair of errors in data structures", *Conf. on Object Oriented Programming Systems Lang. and Applications*, 2003.

[6] M. Ernst, J. Cockrell, W. Griswold et al., "Dynamically discovering likely program invariants to support program evolution", *IEEE Trans. on Software Engineering*, 27(2), 2001.

[7] D. Garlan, S-W. Cheng, B. Schmerl, "Increasing system dependability through architecture-based self-repair", *Architecting Dependable Systems*, 2003.

[8] J. Hollingsworth, B. Miller, M. Goncalves, et al., "MDL: A language and compiler for dynamic program instrumentation", *Conf. on Parallel Architecture and Compilation Techniques*, 1997.

[9] C. Jaramillo, R. Gupta, M. L. Soffa, "FULLDOC: A full reporting debugger for optimized code", *Proc. of Static Analysis Symposium*, 2000.

[10] P. Kessler, "Fast breakpoints: Design and implementation", *ACM SIGPLAN Conf. on Programming Lang. Design and Implementation*, pp. 78–84, 1990.

[11] V. Kiriansky, D. Bruening, S. Amarasinghe, "Secure execution via program shepherding", *USENIX Security Symposium*, August 2002.

[12] J. R. Larus, E. Schnarr, "EEL: Machine-independent executable editing", *Conf. on Programming Lang. Design and Implementation*, June 1995.

[13] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, et al., "The Paradyn parallel performance measurement tools", *IEEE Computer*, 28(11), 1995.

[14] Pin, http://rogue.colorado.edu/Pin/.

[15] ProbeMeister, http://www.objs.com/ProbeMeister.

[16] K. Scott, N. Kumar, S. Veluswamy, B. Childers, J. Davidson, M. L. Soffa, "Reconfigurable and retargetable software dynamic translation", *Int'l. Symp. on Code Generation and Optimization*, 2003.

[17] A. Srivastava, A. Eustace, "ATOM: A system for building customized program analysis tools", *Conf. on Programming Lang. Design and Implementation*, 1994.

[18] A. Zeller, "Isolating cause-effect chains from computer programs", *Int'l. Symp. on the Foundations of Software Engineering*, 2002.