FIST: A Framework for Instrumentation in Software Dynamic Translators

Naveen Kumar, Jonathan Misurda, Bruce R. Childers, and Mary Lou Soffa

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
{naveen, jmisurda, childers, soffa}@cs.pitt.edu

Abstract

Software dynamic translators (SDT) typically monitor, profile, and affect the execution of a program. Such systems have been used to build many useful applications, including dynamic code optimizers for binary machine code and Java bytecode, software security checkers, binary translators, code profilers and program introspection tools. While all of 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 SDT infrastructures. In this paper, we describe such a new 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 demonstrate FIST's flexibility and evaluate it in the Strata SDT system for the SPARC and the Jikes Research Virtual Machine for Java on the Intel x86. To show the framework's scalability, we describe new instrumentation applications to prototype dynamic optimization techniques and apply software tests to Java programs.

1 Introduction

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], 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 *software dynamic translators* (SDT) have been used for enforcing security policies [15], simulating processor architectures [5,20,21,28], just-in-time compilation of Java programs [1], and debugging programs [13,19,25]. All of these systems use information about the executing program to make decisions about how to control the program's execution. For example, program debuggers insert instrumentation into a program to track values [13], security checkers use instrumentation to check for vulnerabilities [15,22], and dynamic optimizers and binary translators use information to identify hot code segments and collect profiles for optimization [2,3,6,8,9,12]. Instrumentation in these and other systems is

used for both information gathering and control or modification of the executing program.

Many techniques and systems have been proposed that use instrumentation to monitor and control a program's execution. These techniques include static binary rewriting of application code for profiling [16,23] and dynamic instrumentation of programs [2,11,18,27]. There have also been infrastructures that aim to provide instrumentation capabilities for different machine platforms [11,16]. However, these system lack a common interface and general framework for instrumentation in SDT infrastructures. Yet, with the importance and number of SDT applications, 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 runtime information gathering and control.

This paper describes **FIST**: a new **F**ramework for program **I**nstrumentation in **S**oftware dynamic **T**ranslation infrastructures that is both scalable and flexible. The framework can gather information, control a program, and dynamically adapt instrumentation. It can operate at different program granularities and map information from one level to another. FIST is also flexible enough to be portable across different infrastructures and machine architectures. It provides 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 software dynamic translator that incorporates FIST.

This paper makes several contributions, including:

- A novel framework (FIST) that combines an event-response model and instrumentation primitives to enable flexible and scalable information collection and control in SDT systems,
- 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 VM and JIT for Java (Jikes) for the Intel x86,
- A use of FIST for a new approach to preloading a dynamic optimizer's trace cache to minimize the opportunity cost and start-up delay of trace formation,

- A use of FIST for a new path-based and very low cost approach for structural testing of Java programs, and
- An evaluation of the performance and memory overhead of our primitives and instrumentation applications in Strata for the SPARC and Jikes for the x86.

Such a flexible and scalable instrumentation framework for SDT can be configured for many applications and uses. For example, it can be used to enforce security policies by monitoring program vulnerabilities [15] and operating system calls [22]. 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 aid optimization in a Java JIT compiler and dynamic optimizer such as Jikes [1]. 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.

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 presents novel applications of FIST to instruction trace formation and structural testing. Section 5 describes related work and Section 6 summarizes the paper.

2 FIST

Our instrumentation mechanism for SDT 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 runtime 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.

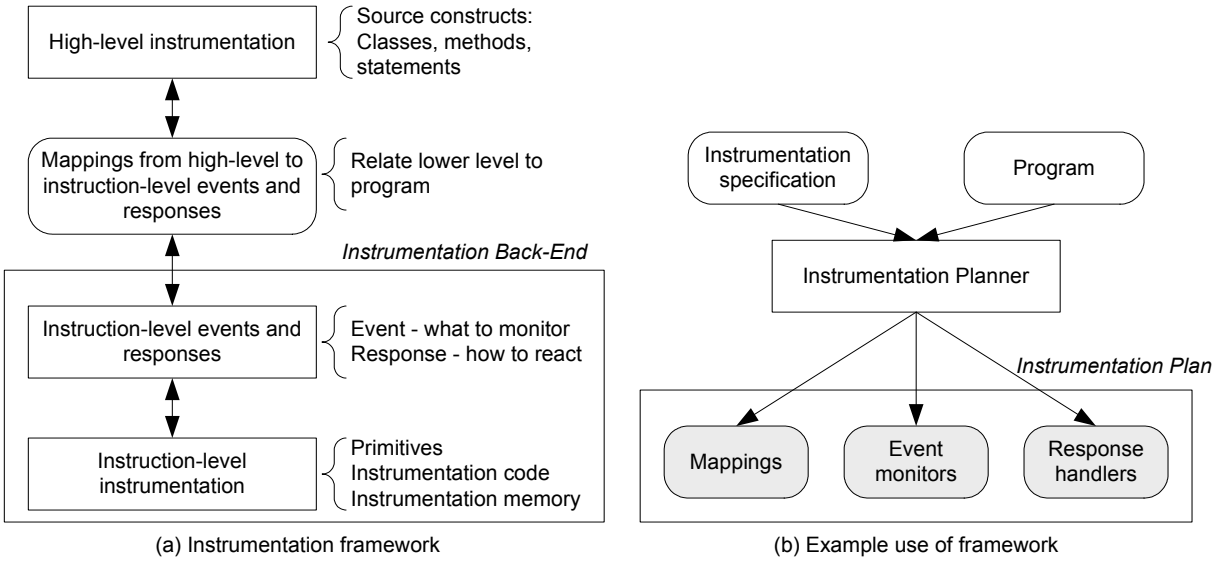


Figure 1: FIST and an example of its use

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 map-

pings 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, optimizing some method, or updating a counter.

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 run-time. 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".

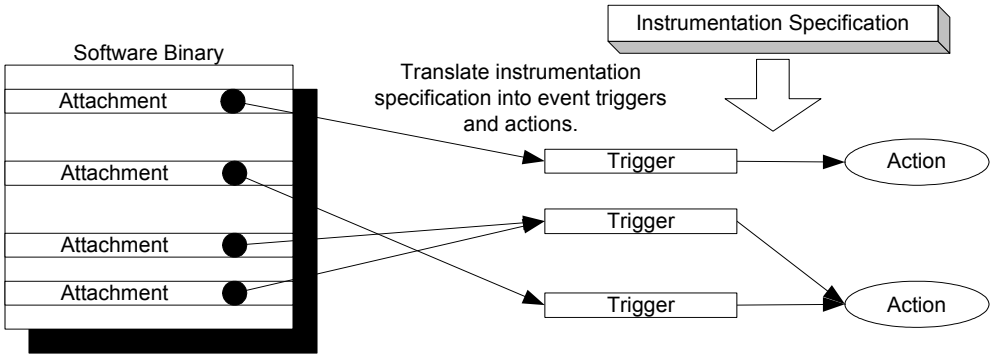


Figure 2: Event-based model for instrumentation with trigger-action pair

Figure 2 shows how the trigger-action mechanism works. In the figure, an instrumentation point is attached to the binary program to transfer 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 (not shown).

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. 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 [14] that takes the flow of

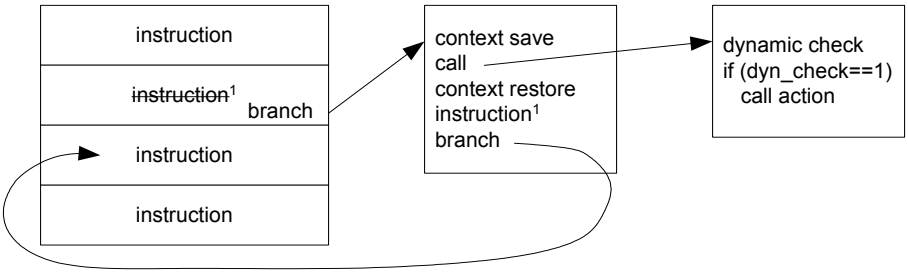


Figure 3: Hit-many for a dynamic check on a fixed-length instruction set

control to code that does the dynamic check. Figure 3 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 in-line 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 block 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 that can be used to hold persistent information and variables needed by the instrumentation itself.

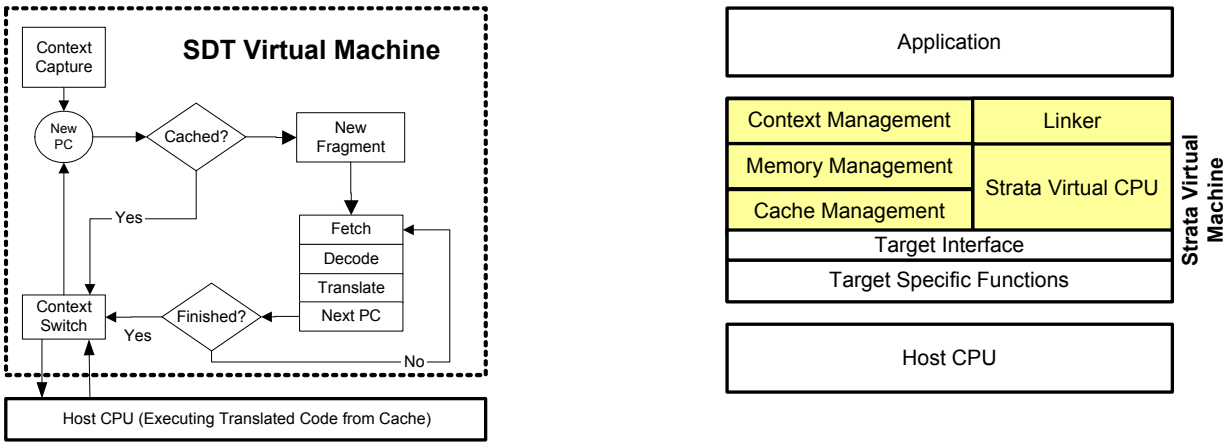
3 Software Dynamic Translation Instrumentation

To demonstrate the flexibility of FIST, we have incorporated it in a SDT for the UltraSPARC 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 systems.

3.1 Strata and SPARC/Solaris

To address the difficulty of building software dynamic translators, we (with the University of Virginia [21]) developed a highly configurable and retargetable SDT infrastructure called Strata. It has been used to implement policies on the use of operating system calls, architectural simulators, and code profilers and it supports several machine platforms. We have integrated our framework in Strata to provide a flexible interface for instrumenting programs in different SDT applications. We first describe how we incorporated FIST in Strata and then describe the unique challenges with the SPARC instruction set.

Strata SDT. Figure 4 shows the structure of Strata, which 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



(a) Strata software dynamic translator

(b) Strata common and target services

Figure 4: Strata virtual machine

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 a set of target-independent common services, a set of 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 to Strata 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 architecture 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.

SPARC ISA. Our primitives handle SPARC V8 and the main challenges were how to implement *hit-once* and *hit-always* for a fixed-length instruction set, the SPARC's branch architecture, and register windows. *Inline-hit-always* has similar challenges as hit-once and hit-always, so we only describe these primitives.

Hit-once Primitive. For *hit-once*, the instruction that is being instrumented is replaced by a jump that transfers control to a breakpoint handler to invoke the dynamic check and action. The breakpoint handler has code for saving and restoring application context and doing the dynamic check. Just before the application context is restored, the handler copies the instrumented instruction back to its original location. After this copy operation, control is transferred to the instruction where the breakpoint was hit. Copying the instruction back and executing it at its original location removes the breakpoint immediately after it has been hit.

Hit-many Primitive. This primitive remains in the program until removed. It replaces an instrumented instruction with a jump and inserts the replaced instruction in the breakpoint handler. In this case, the instrumented instruction executes as part of the breakpoint handler, rather than in the original location of the program. The breakpoint jump remains until it is explicitly removed. To remove *hit-many*, the original instruction(s) from the breakpoint handlers is copied to its original location, overwriting the breakpoint.

Branch Architecture. The SPARC's branch architecture presented a challenge because it has delay slot instructions (DSI). When instrumenting a DSI, the instruction must be replaced by a branch, but the SPARC does not allow a branch to be put in a delay slot. To avoid this problem, we instrument the branch rather than the DSI. In this way, the original branch can be copied back to the program and control transferred to that branch. Such an approach works because the values in the general-purpose registers remain the same while executing either the branch or the DSI. In the case of annulled branches with instrumented DSIs, we test the condition code before invoking the dynamic check to ensure that the check is invoked only if the branch is going to be taken. Another consideration for DSIs occurs when instrumentation is placed at a branch. Here, care must be taken to move the DSI and the original source instruction to the breakpoint handler. The DSI is executed in the delay slot of the branch inside the breakpoint handler to pre-

serve the semantics of the branch. The offset of a PC-relative branch is modified when copied into the breakpoint handler to be the correct taken-target address. The new not-taken target is an unconditional branch that returns control to the fall-thru (not-taken) path of the instrumented branch.

Register Windows. The SPARC has register windows, which made it relatively easy and low cost to implement the context switch for the breakpoints. A SAVE is used to acquire a new set of general-purpose registers and a RESTORE is used to recover the original registers. A breakpoint may also save and restore other registers, including the global and floating point registers, and the condition code and Y register. A partial context switch may be done to save and restore only those registers needed by the dynamic check (and action) to avoid the cost of saving and restoring the global and floating point registers and the Y register.

Experiments. We measured the average memory and performance overhead of *hit-once* and *hit-many* instrumentation 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. These results were collected on a 500 MHz UltraSPARC IIe Sun Blade 100 with 256 MB of RAM and Solaris. To compute performance overhead, we used a program that has a tight loop that iterates 100 million times. The loop body was instrumented on every iteration. Only the instrumentation primitive cost is measured; the overhead of the dynamic check and action are not included because they are application dependent.

	Hit-once	**Hit-many**	**Inline-hit-always**
Time (ns)	660	640	510
Number instructions	72	53	49

Table 1: Memory and performance cost 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 several control transfers. *Hit-once* also does a cache flush when the original instruction is replaced, which has a performance impact. *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.

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. We start the discussion with the x86 ISA, since it impacted the implementation of FIST in Jikes.

x86 ISA. Because the x86 is a variable length ISA, it had special challenges for the implementation of the *hit-once* and *hit-many* primitives, which required a new breakpoint called a "source-sink breakpoint". Two other challenges involved basic blocks smaller than five byte and basic blocks that jump to themselves.

Hit-once and Hit-many Primitives. To implement these primitives, we used a pair of breakpoints called "source-sink breakpoints" to insert and remove instrumentation across two instructions (or basic blocks). In the case of a *hit-many* primitive, source-sink breakpoints insert a breakpoint at the original instruction that is instrumented (the "source"). When hit, the source breakpoint copies the instrumented instruction to its original location to be executed and inserts a breakpoint at the next instruction (the "sink") in the program. When executed, the sink breakpoint re-inserts the original source breakpoint, ensuring that the instrumentation remains until explicitly removed. A difficulty occurs when the next instruction can fall along different paths. We insert sink breakpoints at the possible target blocks, and when one of these sink breakpoints is hit, it removes any sibling breakpoints. Alternatively, we could handle branches by extracting the branch target address from the instruction and putting a sink breakpoint at the target address in the register. *Hit-once* instrumentation is implemented similarly, except the sink breakpoint is not inserted. In this way, when the source breakpoint is hit, it will copy the instrumented instruction to its original location, but the source breakpoint will not be re-inserted.

For a variable length instruction set, copying the instrumented instruction back to its original location works better than executing the instruction in the breakpoint handler. If the instrumented instruction is executed in the breakpoint handler, instructions have to be decoded to find instruction boundaries because an entire instruction must be copied to the handler. Indeed, in some cases, multiple instructions may have to be copied and executed in the handler because the breakpoint jump can span multiple source instructions. Branch target addresses also have to be rewritten when placing a breakpoint at a branch. Our source-sink breakpoints do not know anything about the instructions where the breakpoint is inserted, which simplified and reduced the cost of the instrumentation.

Short Basic Blocks. Another challenge involves the amount of space needed in the block to insert a breakpoint. Our instrumentation system normally uses five byte jumps for breakpoints. However, a basic block can be shorter than five bytes and a breakpoint jump may need to span two successive basic blocks, which can cause a problem when control does not go through the *short block*. The short block problem is determined by how close a breakpoint is placed to the end of a basic block. If the breakpoint is placed closer than five bytes from the end, the short block problem can occur. We handle this problem by putting a breakpoint early in a block to maximize the distance from the block's end. When a breakpoint can not be moved early in a block, the block is padded with space so that the breakpoint can be inserted. Alternatively, the x86's INT 3 instruction can be used to raise a software interrupt to invoke the breakpoint. This instruction takes one byte and always fits in a block, but it is too expensive for general instrumentation [11].

Reflexive Basic Blocks. A final challenge happens for a *reflexive block*, which is a block with a jump to itself. For hit-many instrumentation on the x86, we use source-sink breakpoints; however, these breakpoints can not always handle a reflexive block. Consider the case when the source breakpoint is first hit in a reflexive block. That source breakpoint will attempt to insert a sink breakpoint at its successors, including itself. The sink breakpoint is inserted, and then the source breakpoint copies the original instrumented instruction back to the reflexive block, which may overwrite and remove the sink breakpoint. This case happens when a reflexive block is so short that the sink breakpoint must be placed at the original location. To handle reflexive blocks, we insert a dummy block on the reflexive block's back edge to hold a sink

breakpoint. We also rewrite the branch in the reflexive block to avoid going through the dummy block when the reflexive block is not instrumented. Although this solution changes the program code, it has a minimal performance cost and minimally impacts the application's code storage and layout.

Jikes RVM. To integrate our framework into Jikes, we had to address three issues. The first one was how the instrumentation system gets control to instrument a method, the second one was how to handle multi-threading, and the final one was the interaction of garbage collection (GC) and instrumentation.

Instrumentation Injection. To get control when a method is loaded and compiled, we made a simple modification to the VM to insert a breakpoint in a method's prologue that generates an event whenever a method is entered. The response for that event gets control when a method is entered. 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 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. An alternative is to save information that is persistent within a single method invocation as part of the method's activation frame. When a thread switch occurs, the program stack is switched, which automatically saves the persistent state. Our current implementation uses a combination of these two techniques. Small amounts of information local to a method are stored on the activation frame and large persistent data structures are stored in a separate memory buffer. Putting infor-

mation on the activation frame was a pragmatic decision as it simplified the implementation. The information in the activation frame could be moved to the memory buffer, if desired.

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 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 memory and performance cost of source-sink breakpoints for *hit-once* and *hit-many* instrumentation. The memory cost includes the code size for a breakpoint and storage space to hold instrumented instructions. Because the performance cost of source breakpoints varies with the number of sink breakpoints inserted, we measured the average time of source breakpoint for a benchmark program, 201.compress, from the SPECjvm98. Our experimental platform was Jikes 2.1.1 and a 700 MHz Pentium III dual-processor machine with 512 MB of memory and RedHat Linux 2.4.18.

Description	**Cost**
Source breakpoint execution time	840 ns
Source breakpoint code size	68 bytes
Average storage cost for a source breakpoint	11 bytes

Table 2: Performance and memory cost for breakpoints in Jikes and x86 on 201.compress

Table 2 shows the average cost of a source breakpoint for 201.compress. The performance cost for *hit-many* is approximately twice the value in the table because two breakpoints are executed (source and sink breakpoints). In the breakpoint code, a loop iterates over a list of the successor blocks and inserts sink breakpoints in those successors. On average, there are 1.58 sink breakpoints inserted for each source breakpoint in 201.compress, so the loop overhead is minimal. In our implementation, *hit-once* uses the

same mechanism as *hit-many*, except there are zero successors and the loop that inserts sink breakpoints is not executed. Although the cost of *hit-once* is approximately the cost of the source breakpoint in the table the actual cost is marginally less because the breakpoint does not insert any sink breakpoints (the source breakpoint in the table inserts on average about one and a half sink breakpoints). The table also shows the memory overhead of a breakpoint, which has 25 instructions and takes 68 bytes. There is also on average 11 bytes of data storage that is needed to the hold the original instruction and to store the addresses where sink breakpoints should be placed.

4 Applications

We have used FIST in Strata and Jikes for several purposes, including a code profiler, fast architecture cache simulation, and program debugging. In this section, we describe our approach for two new applications: prototyping for dynamic optimization and structural testing of Java programs to demonstrate our framework's scalability for different amounts and types of information gathering.

4.1 Trace Cache Management

The flexibility and scalability of FIST in Strata makes it an ideal tool for exploring the benefits of new techniques for dynamic code optimizers, and we used FIST to investigate an approach for forming instruction traces off-line for dynamic optimization. Dynamic optimizers often use instruction traces, composed of basic blocks along a hot path, as the granularity for optimization, and typically these traces are collected online so they match a program's dynamic behavior. There is an opportunity cost associated with identifying these traces because past history must be collected to identify candidate traces. However, work by Hazelwood and Smith shows that traces do not have much variability in certain circumstances [10]. This observation motivates the idea of forming traces off-line to pre-load or seed dynamic trace formation. Forming off-line traces help reduce the opportunity cost of identifying traces online and more powerful and expensive techniques can be used to find traces. The disadvantage is that off-line traces may not match actual behavior, particularly when input data sets have a large influence on a program's execution.

We prototyped an off-line trace formation technique to preload the fragment cache. FIST was used to develop an instrumentation planner that could both profile an application and investigate the benefits of

our off-line trace formation technique. The instrumentation planner uses an algorithm that we call "next heaviest edge" (NHE) to determine the traces to be preloaded. NHE forms traces by starting with a seed edge from a profile that has the heaviest weight and the blocks associated with this edge are added to a new trace. NHE adds new blocks to the trace by selecting the successor and predecessor edges with the heaviest weight until an end of trace condition is encountered. The end of trace condition takes into account the significance of successor and predecessor blocks, edge execution frequencies, amount of code duplication, and the size of a trace. NHE is useful because it captures characteristics of other trace formation algorithms, such as Next Executing Tail [8], and can be used to explore the benefit of different algorithms.

To form traces off-line, we needed to instrument the code in Strata's fragment cache in different ways to collect profile information and run-time statistics. Using our instrumentation primitives, we built different kinds of profile operations, including *counter* and *sampler* operations. A counter records the number of times an edge is executed and can be implemented by *hit-many* instrumentation that increments a counter when an edge is hit. A *sampler* can be implemented similarly, except it can control the insertion and removal of instrumentation based on the sample count. A *counter* and a *sampler* can be combined to control the overhead of gathering edge profiles.

Experiments. To determine the benefit of NHE, we did experiments with Strata on the SPARC (see Section 3.1) and evaluated NHE with some SPEC2000. For each benchmark, we did two runs. The first run profiled the benchmarks with the train input from SPEC2000. We used NHE with these profiles to compute traces off-line. On the second run, the traces were pre-loaded into Strata's fragment cache, and using the reference input, each benchmark was run to determine the benefit of NHE. Figure 5 shows the percentage performance improvement of preloading traces with Strata over not preloading the traces with Strata. The cost of preloading the fragment cache was minimal in comparison to program run-time. Forming traces improved performance from 6% to 35%, but in one case, *mcf*, there is a negligible slowdown.

FIST was used to investigate why *mcf* had a performance slowdown and we found that traces are exited early 71% of the time. Hence, the traces formed with the training data were not representative of those needed by the reference data and shows a limitation of off-line trace formation. Using the SPARC's

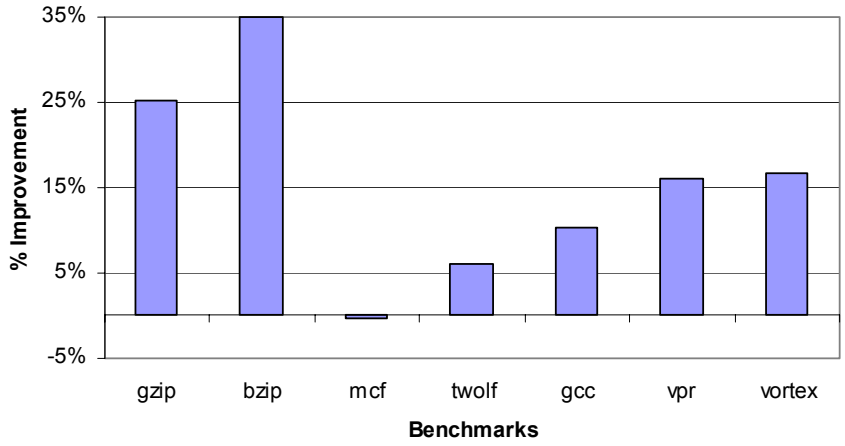


Figure 5: Performance improvement with a preloaded fragment cache in Strata

performance counters, we found that *mcf* with preloaded traces had a large number of instruction cache misses and branch mispredictions.

From these experiments, we believe overall there is a benefit to preloading traces and we are continuing work on improving NHE and developing other ways to convey static hints to a dynamic optimizer. FIST let us quickly prototype this idea—it took only three days to implement, debug, and run experiments. This application shows the benefit of flexible instrumentation.

4.2 Branch Coverage Testing

Using FIST in Jikes, we are developing a new method for structural testing and profiling of Java programs. Our approach uses a novel scheme to employ test and profile plans that describe what instrumentation should be automatically inserted and removed in executing code to carry out testing or profiling strategies. A test or profile plan is a recipe that describes how and where instrumentation should be performed. The approach is path specific and uses the actual execution paths of an application to drive the instrumentation. Once instrumentation is no longer needed at a particular point, it is dynamically removed to avoid run-time performance overhead, and the test/profile plan continues.

We used FIST to explore the benefit of our path-based approach. We have developed an instrumentation planner (see Figure 1b) for branch coverage analysis that generates test plans using our primitives. Branch coverage is a structural test that determines which edges in a program have been executed. The main function of the test planner is to determine where and how to insert instrumentation and to generate a test plan that consists of test data and instrumentation actions. The test data has information about

how to do branch coverage and says when to insert and remove instrumentation for covering each edge in a method. The action is target machine code that is executed at each instrumentation point to update a table that records which edges have been covered. The action also removes instrumentation once an edge has been covered and inserts new instrumentation to cover edges that are next to execute and have not yet been hit. In this way, the action inserts and removes instrumentation dynamically along a path of execution.

Experiments. Using FIST and our test planner, we built a tool for branch coverage testing with Jikes. The tool lets a user graphically specify what code regions to apply branch coverage testing on and a coverage criteria. It generates a textual specification that is used by the test planner to determine a test plan.

Benchmark	Overhead	Benchmark	Overhead	Benchmark	Overhead
201.compress	1.2%	227.mtrt	11%	222.mpegaudio	2.2%
209.db	2.5%	202.jess	4.8%	213.javac	9.2%

Table 3: Run-time overhead for branch coverage

We have done experiments to determine the overhead of our branch coverage tool built with FIST. Table 3 shows the run-time performance overhead for several benchmarks relative to running the same programs without applying branch coverage. The benchmarks were run on the experimental platform described in Section 3.2. The overhead includes the time to construct an instrumentation plan and execute the instrumentation inserted in the code to determine coverage. As the table shows, run-time overhead is minimal, varying from 1.2% to 11%. Interestingly, in some cases, we found that the test planner overhead was more than the instrumentation overhead. This case can occur when a method has many basic blocks, but a short path is taken through the method. In general, however, across all methods, the test planner accounts for a very small portion of the run-time overhead and the majority is spent in the instrumentation code. The overhead of the instrumentation is quite small because the instrumentation is immediately removed once an edge is covered. The flexibility of our instrumentation approach made it possible to build such an end-to-end branch coverage tool that has very low overhead.

5 Related Work

Instrumentation techniques have been used in software dynamic translators for a number of purposes, including dynamic optimization [1,2,3], program debuggers [13,19,25], software security [15,22], and

binary translation [6,7,9]. In all of these systems, the instrumentation is hard coded into the system. In Dynamo [3], the instrumentation happens in the interpreter and in Dynamo/RIO, instrumentation is typically inserted on back edges inside a basic block [4].

The concept of fast breakpoints was pioneered by Kessler [14], which used the technique that we call *hit-many* instrumentation. Kessler's fast breakpoints were not applied in a flexible manner across different SDT and architectures to dynamically instrument programs. Code modification and instrumentation systems like Vulcan [24], Dyninst [11] and Paradyn [18] used a technique similar to ours to instrument a program with fast breakpoints. Like our framework, Dyninst is intended to be general, with a language for specifying instrumentation [11]. Both Dyninst and Paradyn were built for very specific purposes, and to the best of our knowledge, their instrumentation techniques were not designed to support different SDT infrastructures. Originally, Dyninst was able to only instrument at the method level, but it has recently been extended to instrument programs at the instruction-level [17]. However, a general framework, primitives and implementation for different software dynamic translators with Dyninst has not been described.

Dyninst has been used to build a software testing tool that is similar to our testing tool [26]. The Dyninst tool dynamically inserts instrumentation on method invocations for node coverage, where we take a similar approach for branch coverage (a stricter form of coverage). Unlike our approach, instrumentation is inserted in the whole method when it is invoked and a separate garbage collection process is done to remove instrumentation. Our technique instruments only along executed paths and removes instrumentation on-demand as soon as possible to keep run-time overhead minimal.

Other work with dynamic instrumentation includes *ephemeral instrumentation* that uses lightweight instrumentation for program profiling [27]. This approach gathers branch biases and constructs an edge profile by post processing sampled profile information. The paper, however does not provide a general infrastructure for instrumentation that can, for instance, be used at run-time by dynamic optimizers or for persistent instrumentation by architectural simulators like Shade [5]. FIST, however, can be easily configured to perform such lightweight ephemeral instrumentation in different SDTs.

Arnold and Ryder proposed a scheme that inserts counters into a program to guard the execution of instrumented code [2]. With our *inline-hit-always* primitive and a counter, we can implement similar

instrumentation using code duplication. To avoid code duplication, we can also use *hit-many* and *hit-once* to insert and remove instrumentation dynamically based on a counter. This approach also avoids disturbing the application with inline instrumentation; however, it does increase run-time cost of the instrumentation.

6 Summary

This paper described a framework for flexible and scalable instrumentation, called FIST, in software dynamic translators, such as Strata and Jikes. We demonstrated FIST's capabilities on two different SDT infrastructures, two target architectures, and two instrumentation applications. 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. We used our infrastructure in Strata to prototype a technique to preload and seed a trace cache of a dynamic optimizer to reduce the cost of dynamically discovering instruction traces. We also showed that FIST can be used for structural testing of Java programs. These framework instances and applications show that our approach is indeed flexible and scalable.

7 References

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

[2] M. Arnold and B. G. Ryder, "A framework for reducing the cost of instrumented code", *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 2001.

[3] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system", *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 2000.

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

[5] R. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling", Technical Report 93–06–06, Computer Science, University of Washington, June 1993.

[6] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing Software: Using speculation, recovery, and adaptive retranslation to address real-life challenges", *Int'l. Symp. on Code Optimization and Generation*, March 2003.

[7] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. Fisher, "DELI: A new runtime control point", *Int'l. Symp. on Microarchitecture (MICRO-35)*, November 2002.

[8] E. Duesterwald and V. Bala, "Software profiling for hot path prediction: Less is more", Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems, 2000.

[9] K. Ebcioglu and E. Altman, "DAISY: Dynamic compilation for 100% architectural compatibility", *24th Annual International Symposium on Computer Architecture (ISCA'97)*, June 1997.

[10] K. Hazelwood and M. D. Smith, "Characterizing Inter-Execution and Application Optimization Persistence", *Workshop on Exploring the Trace Space for Dynamic Optimization Techniques* held in conjunction with the *17th Int'l. Conference on Supercomputing*, San Francisco, CA, June 2003.

[11] 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.

[12] R. Hookway and M. Herdeg, "DIGITAL FX!32: Combining Emulation and Binary Translation", *Digital Technical Journal,* Vol. 9, No. 1, August 1997.

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

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

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

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

[17] J. Marathe and F. Mueller, "Detecting memory performance bottlenecks via binary rewriting", *Workshop on Binary Translation*, during *Conf. on Parallel Architecture and Compilation Techniques*, Sept. 2001.

[18] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, et al., "The Paradyn parallel performance measurement tools", *IEEE Computer*, Vol. 28, No. 11, November 1995.

[19] N. Ramsey and D. Hanson, "A retargetable debugger", *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 1992.

[20] E. Schnarr, *Applying Programming Language Implementation Techniques to Processor Simulation*, PhD thesis, University of Wisconsin, Madison, 2000.

[21] 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*, March 2003.

[22] K. Scott and J. Davidson, "Safe virtual execution using software dynamic translation", *2002 Annual Computer Security Applications Conference*, Dec. 2002.

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

[24] A. Srivastava and A. Edwards, "Vulcan: Binary transformation in a distributed environment", Microsoft Research Technical Report, MSR–TR–2001–50, April 2001.

[25] R. M. Stallman and R. H. Pesch, "Using GDB: A guide to the GNU source-level debugger", GDB version 4.0. Technical report, Free Software Foundation, Cambridge, MA, 1991.

[26] M. Tikir and J. Hollingsworth, "Efficient instrumentation for code coverage testing", *Int'l. Symp. on Software Testing and Analysis (ISSTA)*, June 2002.

[27] O. Traub, S. Schechter, and M. D. Smith, "Ephemeral instrumentation for lightweight program profiling", Technical Report, Harvard University, 2000.

[28] E. Witchel and M. Rosenblum, "Embra: Fast and flexible machine simulation", *Conf. on Measurement and Modeling of Computer Systems*, May 1996.