Flexible Instrumentation for Software Dynamic Translation

Naveen Kumar and Bruce Childers

Department of Computer Science
University of Pittsburgh
{naveen, childers}@cs.pitt.edu
http://www.cs.pitt.edu/copa

Abstract

Software dynamic translators have been used for many purposes, such as dynamic code optimization, profiling, and security. Many of these applications need to instrument a program's binary code to gather run-time information about the program. Such instrumentation is varied, with different software dynamic translator applications requiring different kinds of information. Hence, there is a need for a flexible mechanism for information gathering and instrumentation in software dynamic translators. In this paper, we describe our approach to providing flexible instrumentation. We also experimentally evaluate our approach and investigate its overhead and demonstrate its flexibility for different software dynamic translation systems.

1. Introduction

In recent years, there have been a number of systems introduced that dynamically modify and control the execution of a program. For example, dynamic optimizers such as Jalapeno [4] and Dynamo [3] apply code transformations to take advantage of up-to-date information about the run-time behavior of the program. Other examples of such *software dynamic translators* (SDT) include code security checkers [2, 8], micro-architecture simulators [13, 14], and program debuggers [19]. SDT systems typically collect information about the executing program in order to make decisions about how to control the program's execution. For example, program debuggers may insert instrumentation in the application binary to collect information about program values. As another example, dynamic code optimizers generally focus their optimization efforts on the most frequently executed portions of the program, which requires instrumentation to identify hot code segments and traces. In Dynamo, instrumentation is used for counters that help identify hot traces for optimization. Similarly, Jalapeno uses counters to find hot methods for optimization.

To support diverse instrumentation needs in software dynamic translators, a flexible mechanism is needed for instrumenting a program at run-time. Such a mechanism can be configured for a variety of purposes, including software testing, dynamic optimization, program profiling, and architecture simulation. There are many considerations when designing a mechanism for code instrumentation, including run-time performance and memory overhead, flexibility to gather different kinds of information and different durations of time, ability to dynamically insert and remove instrumentation, and support different granularities of information gathering. We discuss each of these requirements below.

First, the instrumentation should have low cost. In an SDT system, the instrumentation occurs during the run-time of the program and its cost must be kept low. Also, same instrumentation might be invoked several times when the code is executed. It is important to have a low overhead of the instrumentation, in terms of number of instructions.

Second, different applications need to gather different kinds of information by instrumenting the code; hence the instrumentation system should be easily reconfigured for different purposes. Also, the instrumentation should be done in a manner that is independent of the target architecture to aid in retargeting an SDT system to new processors and operating systems.

Third, the system should be flexible enough to allow for different life spans of instrumentation. Instrumentation that remains in place throughout the execution of the program is *permanent instrumentation*. Such instrumentation could be needed, for instance, when profiling the program to collect edge counts. In this case, the run-time overhead of the instrumentation is more important than the static cost of inserting the instrumentation. Instrumentation that remains for some portion of the execution of the program is *transient instrumentation*. Transient instrumentation needs to be removed after some period of time during the execution of the program. Such instrumentation could be needed, for instance, when doing path coverage analysis, or when sampling a program at certain intervals during execution. For such instrumentation, the static overhead of inserting the instrumentation may be as important as the run-time overhead.

Fourth, an instrumentation mechanism should provide for low cost dynamic insertion and removal of

instrumentation. The mechanism should permit inserting instrumentation into the executing program to gather new information at new locations in the program. Similarly, instrumentation should be removable on-the-fly as the program executes. To aid in the insertion and removal of instrumentation, the instrumentation should disturb the binary program as little as possible. By minimally impacting the program, it will be easier to quickly insert instrumentation and recover the original form of the program when removing instrumentation.

Finally, it should be possible to have several levels of instrumentation at the same point in the binary program. For example, if the goal is to be able to profile the program and collect all edge counts and all function-call counts, we may need to instrument a single call instruction twice, for both purposes. The instrumentation mechanism should also permit different granularities of information gathering. These levels may include individual instructions, basic blocks, code traces, call sites, and other levels.

This paper describes a flexible instrumentation mechanism that satisfies the above requirements. Our approach uses a trigger-action mechanism for instrumentation. A *trigger* is fired when some property of the program is satisfied and an *action* is taken when a trigger is fired. As an example, consider instrumentation for code security checks of system calls in a software dynamic translator. For system calls, there can be unsafe conditions (determined by the security policy being enforced) when the system would take some appropriate action like terminating the execution of the program. For this purpose, when using our mechanism, we can instrument and associate a trigger with each of these system calls in the program. When the trigger is hit, it executes code that checks whether the system call is safe. If it is unsafe, the trigger is fired and an action routine is called to enforce the security policy.

In this paper, we describe our trigger-action mechanism and an implementation of the mechanism for the SPARC architecture and the Solaris 8 operating system. We also describe the use of the mechanism for applications that gather varying kinds of information to demonstrate the flexibility of our approach. We consider three applications: a hardware cache simulator, a profiler to gather edge counts, and a profiler to find the working set of a program's basic blocks. The overhead of the instrumentation in these applications is evaluated to investigate the run-time overhead of our approach.

The organization of the paper is as follows. Section 2 describes the software dynamic translator, Strata, that is used in this work. Section 3 presents our trigger-action mechanism, while Section 4 discusses the implementation of that mechanism. Section 5 presents experimental results. Section 6 describes related work and Section 7 summarizes the paper.

2. Software Dynamic Translation

SDT allows modification of an executing program by inserting new code, modifying some existing code or controlling the execution of the program in some way. The organization of an SDT system typically has a software layer below the executable that takes control of and modifies the application code. In addition, there is usually a code-cache in which the SDT keeps the modified executable (which we call a *fragment cache*). A program directly executes within the fragment cache after being modified.

SDT works by translating blocks of instructions from the executable program and caching the blocks (after possibly modifying the instructions) in the fragment cache. The blocks of instructions in the fragment cache are called *fragments*. A fragment is a set of instructions that begin at a start-of-fragment condition (a target of a branch or jump) and end with an end-of-fragment condition (a branch or a jump). The branch instruction ending a fragment is modified to branch to an exit stub that returns control to the SDT. The SDT then translates and caches the target fragment into the fragment cache. Once a fragment and its successors are inside the fragment cache, the SDT links them together to avoid unnecessary context-switches between the SDT and the application. In this way, once a program and its fragments have materialized in the fragment cache, execution is entirely out of the fragment cache.

It is possible to modify the instructions during the translation phase based on some criteria and then emit the modified instructions into the fragment cache. This gives the flexibility of an interpreter although the program undergoes direct execution. Such modifications often include code optimizations like partial function inlining or path-based redundancy elimination.

Another common technique used by software dynamic translators is to form *instruction traces*. A trace is a sequence of instructions on a path. A trace collected on a hot path and emitted into the fragment cache has the potential to boost the performance of the program. Such hot paths dominate program execution time and applying optimizations such as code re-layout can significantly improve performance. Traces are important beyond optimization. For example, in applications of SDT that need to collect some information from the running program, collecting the same information from hot traces may be representative of the whole of the program.

In this work, we use the Strata software dynamic translator system [1, 17], which is a retargetable and reconfigurable system. Our work aims to provide a flexible interface for instrumentation in Strata. Strata is implemented as a set of target-independent common services, a set of target-dependent specific services and

an interface through which the two communicate. The common services in Strata include memory management, code cache management, a dynamic linker, and a virtual CPU that mimics the standard hardware (fetch/decode/execute engines). The target-specific services are the ones that actually do the dynamic translation.

Strata is designed as virtual machine that sits between the program and the CPU and translates a program's instructions before they execute on the CPU. The Strata VM is started by a function call from the application binary which saves the application context and starts translating and caching the instructions in the application. Such an infrastructure provides for flexible instrumentation of the application before it materializes in the fragment cache.

Figure 1 shows the working of the Strata virtual machine. In this figure, every instruction that is not already cached goes through different stages in the virtual machine, namely fetch, decode and translate. During the translate stage, Strata can instrument the application code and then write it into the fragment cache. In this work, we provide an interface in the VM for different mechanisms for instrumenting the code. This interface is an extension of the original translate stage, as shown in the figure.

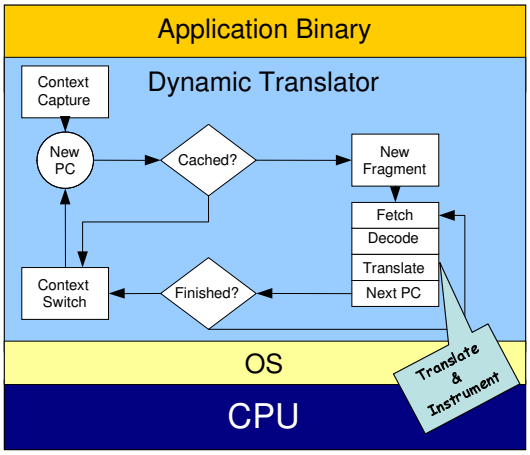

Figure 1: Working of the Strata virtual machine

3. Trigger-action Mechanism

The basic idea of our trigger-action approach is to use an event-driven mechanism that checks for some run-time property of a program. When that property is identified, a call back is made to an action that can perform information gathering and other functions. With our trigger-action mechanism, we can instrument a program at any point in the code and at any time during the program's execution.

The instrumentation inserted into the program to perform the property check and call back are called a "trigger-action pair". The trigger contains a *code property check* that can invoke an associated action, if the property check is satisfied. For instance, in the example mentioned in Section 1, a property check would test an instruction to check whether it is a system call and whether the system call is safe given some security policy on the use of system calls. In this case, for termination semantics on a security violation, the action would abort the program's execution with an access violation.

There are two parts to a trigger: a static component and a dynamic component. The *static* component is a check that can be done by an SDT system, like *Strata,* when writing instructions into the fragment cache. The *dynamic* component verifies a dynamic property of the code. For example, suppose we want to count all addition instructions that have a register operand with a zero value. In this case, the trigger will verify that an instruction is an addition and that the content of one of its source registers is zero. If these conditions are satisfied, the action increments a counter that counts the number of dynamically executed additions with source operand value of zero. In this case, *Strata* can verify that the opcode for a given instruction is an ADD or ADDI with a static check when writing instructions into the fragment cache. If the instruction is an addition, a *dynamic check* is inserted to verify that the content of one of the source operands is zero. Essentially, when attaching triggers to particular code blocks, the instrumentation system will verify the static properties before instrumenting the code. That is, inserting a new trigger-action pair is guarded by a *static check*. The *dynamic check* is done in the code that is inserted.

Figure 2 shows an operational view of the trigger-action mechanism. The left part of the figure shows a fragment cache that has been instrumented at several points. Each of these instrumentation points takes program control to one of the dynamic checks shown in the middle. Each dynamic check can transfer program control to one of the actions shown on the right side.

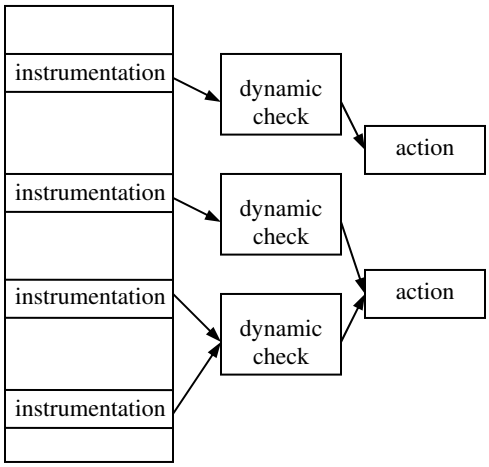

Figure 2: Operational view of the trigger-action mechanism

Notice in the figure that several dynamic checks can be shared by different instrumentation points in the cache. Likewise, there can be several actions that can be shared by different dynamic checks. Such sharing helps to reduce the memory cost of the triggers and actions.

Although it is not shown in the figure, a single instrumentation point could invoke several dynamic checks and their corresponding actions. We call such checks *compound dynamic checks*, since they essentially combine several individual checks into one check that does multiple things. These compound checks help to reduce the performance and memory overhead of doing several checks and actions at some point in the code.

4. Trigger-Action Mechanism for Strata-SPARC

We have implemented our trigger-action mechanism on the SPARC platform using *Strata* [1, 17]. We first describe the basic approach and implementation of the static and dynamic checks for Strata-SPARC. Then we describe three different implementations of the dynamic check. Finally, we describe the implementation of the action.

4.1 Implementing the Static and Dynamic Checks

The static check is easy to implement in *Strata* because of the way the *Strata VM* is organized. We can add static checks to the translate stage in *Strata* so that they are done before any code is written into the fragment cache. When the translated code satisfies the static property being checked, the code is instrumented for the dynamic check (and the action). This instrumentation is performed using a *fast breakpoint*. A fast breakpoint [16] replaces an instruction by a jump instruction which takes the flow of control to a piece of code that may monitor or modify the state of the machine or the program.

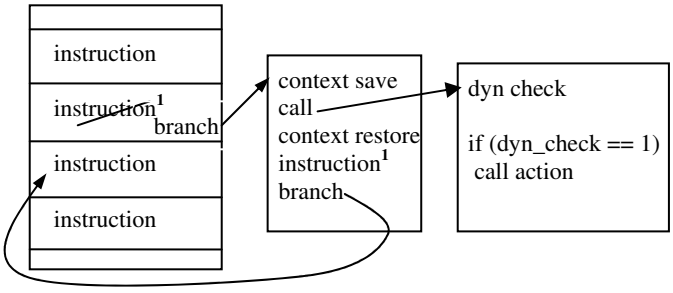

Figure 3: Dynamic check with a fast breakpoint

Figure 3 shows how we use fast breakpoints for dynamic checks. As the figure shows, the code for the breakpoint (the *breakpoint handler*) consists of instructions that save the context of the application, make a call to a boolean function to do the dynamic check, restore the context of the application, execute the original instruction and then jump to the next instruction to be executed in the fragment cache. The context of the application consists of the set of general purpose registers and other machine registers such as the condition code registers and y registers on SPARC. We need to save the context of the application before invoking the dynamic check/action, so that we do not modify any part of the application's context from within the dynamic check/action.

We associate a unique breakpoint handler for every breakpoint because each breakpoint handler has to execute a unique instruction from the application (e.g., instruction[1] in Figure 3) and return to a distinct location in the fragment cache (which is unique to every breakpoint). Our system allocates space in the fragment cache to hold the breakpoint handler. This space is typically located immediately after the fragment. That is, when fragments are created and breakpoints inserted, the handlers are inserted at the end of the fragment being instrumented. Likewise, when adding dynamic checks on-the-fly into already existing fragments, space is allocated in the fragment cache to hold the breakpoint handler. To preserve the code layout of the fragment cache, the breakpoint handlers can be emitted into a separate code cache.

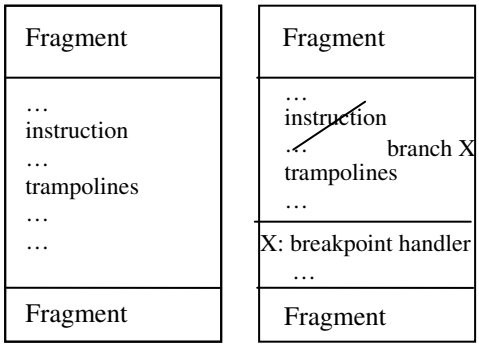

Figure 4: Fragments in the fragment cache without instrumentation (a) and with instrumentation (b). The middle fragment is being instrumented.

Figure 4(a) and 4(b) shows the structure of fragments in the fragment cache without and with the breakpoint. We insert the breakpoint handler at the end of each fragment as shown in Figure 4(b).

For instrumentation techniques that modify the code in the fragment cache, we have to ensure that the machine's data and instruction caches are consistent (in a way similar to self modifying code). On some architectures (SPARC and MIPS), we have to flush a portion (or all) of the instruction cache. On other architectures (Intel's x86), the hardware provides mechanisms that enforce consistency between data and instruction caches.

4.2 Types of Instrumentation

To support different types of information gathering, our trigger-action mechanism has different implementations

of dynamic checks. These implementations differ in the way in which instrumentation is left in place in the application and removed. *Transient instrumentation* removes a dynamic check as soon as the dynamic check is executed ("hit"), while *permanent instrumentation* leaves the dynamic check in place until it is explicitly removed by some action. *Coupled instrumentation* inserts and removes dynamic checks across two fragments in the fragment cache. Each of these implementations are described below.

Transient instrumentation: Figure 5 shows this kind of instrumentation. For this kind of instrumentation we replace the instruction that is to be instrumented by a jump instruction that transfers control to the breakpoint handler. The breakpoint handler has code for saving the context and conducting the dynamic check. Just before the application context is restored, the breakpoint handler replaces the instruction back in its original location. After this, control is transferred to the instruction where the breakpoint was hit. If the breakpoint is implemented in this way, it removes itself after one hit. We can use this approach for transient breakpoints that need to be removed immediately. This approach is inexpensive compared to the other approach in which we need to remove the breakpoints explicitly (discussed below). In our current implementation, the breakpoint code remains unless there is a flush of the fragment cache.

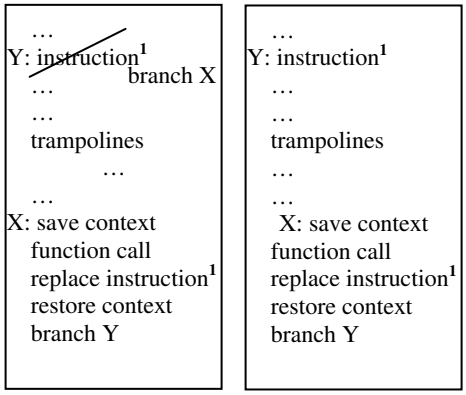

Figure 5: A fragment with transient instrumentation before the breakpoint has been hit (on the left) and the same fragment *after* the breakpoint has been hit.

Permanent instrumentation: Figure 3 shows this implementation. This approach is similar to the previous one, except that we execute the instruction displaced during instrumentation in the breakpoint handler itself.

Figure 6 shows the implementation for the case when the instrumented instruction is a branch. In this case, we move the delay slot instruction to the breakpoint handler as well. Thus, we execute the delay slot instruction in the delay slot of the original branch instruction inside the breakpoint handler to preserve the semantics of the branch. The offset of PC-relative branch instructions are modified when copied into the breakpoint handler to the correct *taken target* location. The new *not-taken target* (the instruction following the instruction in the delay slot) is an unconditional branch instruction that branches to the instruction following the original delay slot instruction in the fragment. This target is the original *not-taken target.*

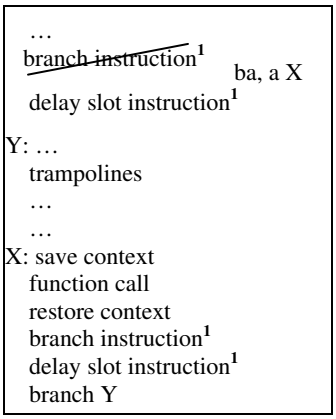

Figure 6: Permanent instrumentation when the instrumented instruction is a branch.

If we need to install a breakpoint at an instruction in the delay slot of a branch, we instrument the branch itself. We benefit from the fact that values in the general purpose registers remain the same while executing either the branch instruction or the instruction in the delay slot. In case of annulled branches, we look at the condition codes before invoking the dynamic check. The dynamic check is only invoked if the branch is going to be taken

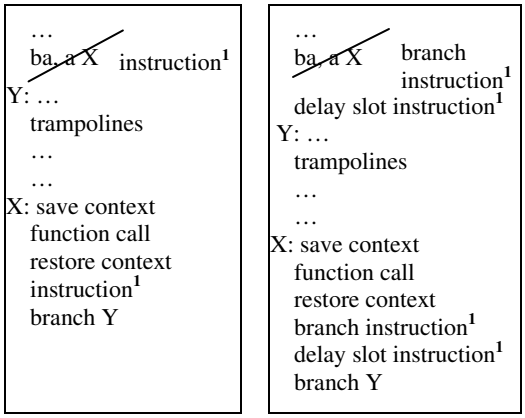

Figure 7: The removal of a breakpoint when the instrumented instruction was a non-branch instruction (left) and the same when it was a branch instruction (right).

Removing the instrumentation involves copying back the instruction to its original location in the fragment cache as shown in Figure 7. The removal of instrumentation is easy because in the code for the breakpoint handler we keep the original instruction at the same offset from the

start of the code. In the case when the original instruction is a branch, the instrumentation moves the delay slot instruction to the breakpoint handler as well. However, we still need to move only the branch instruction. This is because the control transfer, to the breakpoint handler, happens by means of a *branch-always-annulled* instruction. So, the delay slot instruction is duplicated (it is present in the fragment at its original location and inside the breakpoint handler). The instruction in the delay slot of a *branch-always-annulled* never gets executed, so this duplication does not have any computational effect. Figure 7(a) shows the case when the instrumented instruction is not a branch and Figure 7(b) shows the case when it is a branch instruction.

Coupled instrumentation: This type of instrumentation is a combination of the two approaches above and is shown in Figure 8. The figure shows that we replace an instruction by a jump instruction, taking control to the breakpoint handler. The breakpoint handler contains code that copies back the original instruction, but instruments the following instruction in the fragment to take control to another breakpoint handler. Once control reaches the other breakpoint handler, it simply re-installs the first breakpoint and removes the current one. The advantage of this approach is that it is easier to remove a breakpoint once it has been placed. All that is needed to be done is that the second breakpoint handler should not re-instrument the first instruction.

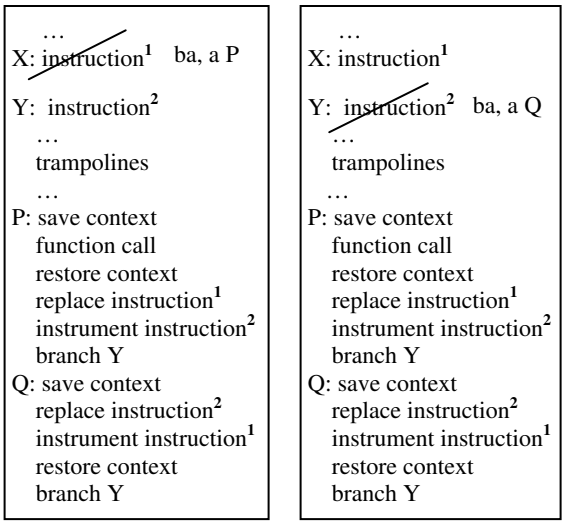

Figure 8: A fragment with coupled instrumentation. The breakpoint handlers are located at P and Q. The fragment before the breakpoints are hit is shown on the left. The same fragment when the breakpoint at location X has been hit; at this time, the breakpoint handler 1 has instrumented instruction^2. After both the hits of the breakpoints, the fragment looks like the one on the left.

However, when a branch instruction is to be instrumented, this technique is more complicated. In this case, the first breakpoint handler needs to instrument the instruction in the delay slot, and because we insert the fast breakpoint at the branch for delay slot instructions, we would instrument the same branch instruction again. To avoid this problem, we instrument the target of the branch. When the branch is hit and the breakpoint is taken, we know the target of the branch and can place the second breakpoint at the target instruction. When the second breakpoint is taken, its breakpoint handler re-inserts the first breakpoint and removes the current one. Hence, the instrumentation is applied across two different fragments in contrast to the previous two techniques.

If the instrumentation needs to remain for a substantial number of hits (or is permanent), the cost of this technique is more than that of the permanent breakpoints. This technique involves twice the number of context saves and restores than the permanent instrumentation.

4.3 Implementing the Action

The *action* is a high-level routine that the system should call when the trigger fires. It is implemented in an architecture independent fashion as a high level function that is called when the dynamic check is satisfied. There is only one instance of the action, although it may be shared by multiple dynamic or compound checks.

The action function is called in the context of the dynamic check. The dynamic check saves the entire application context before invoking the action and restores it before transferring control back to the application. Hence, the action is guaranteed not to affect the context of the executing program. The default behavior of the action is to return to the dynamic check.

5. Experiments

We implemented our trigger-action mechanism in Strata-SPARC and three different uses of our technique. First, we implemented a hardware cache simulator that simulates the instruction and data cache. Second, we implemented a profiler that collects edge counts through the execution of the program. Third, we implemented a system that collects the working set of a program.

Our experiments were run on a lightly loaded 500 MHz UltraSparc IIe workstation with 256 MB of RAM, running Solaris. We measured the memory and CPU overhead of the trigger-action system. To measure the cost incurred by our system in a real application, we compared the performance of the SPEC2000 benchmarks with instrumentation and the performance of running them with Strata-SPARC without instrumentation.

Cost of instrumentation: All the instrumentation techniques have overhead to save and restore the context of the application. They also have the function call overhead for the triggers and the actions. To compute the

memory overhead of instrumentation, we counted the number of instructions required per instrumentation point for each of the three techniques.

To compute the CPU overhead, we wrote a program with a tightly-bound loop iterating for 100 million times and instrumented each of the fragments in the program exactly once. For the cost of instrumentation, we do not count the time taken inside the triggers and actions, since they are application dependent. Hence, we do not have any checks at the trigger, and hence no action is invoked. We used all three instrumentation techniques and measured the run-time overhead of an individual instrumentation point.

In the case of transient instrumentation, instrumentation was removed after one hit of the breakpoint. To measure the cost of the instrumentation, we did not link the fragments in the fragment cache of Strata. This ensures that Strata gains control of the program after each fragment is executed and can re-instrument every fragment before it is executed. We made sure not to link the fragments in the un-instrumented program as well, while computing the overhead. The results of this experiment are shown in Table 1.

	Num. Instructions	Time
Transient	71	660 ns
Permanent	53	640ns
Coupled	166	840ns

Table 1: Memory and CPU overhead of the three instrumentation techniques

Most of the expense of the instrumentation comes from the overhead of saving and restoring the program context. A save or a restore involves 21 instructions each and the overhead of a dummy call to the trigger and action is 7 instructions. The control transfers to and from the breakpoint handler take us 4 instructions and the cost of emitting code at run-time (for transient and coupled instrumentation techniques) is 14 instructions for the first instruction and 5 for each additional instruction. When doing the transient and coupled instrumentation, we also have to flush the machine cache.

Although the SPARC has register windows that can save and restore 24 registers with one instruction, the context switch must save the global registers and some machine registers like the condition code and y registers. One possible way to improve the context switch performance is to do a partial context save and restore, if the registers needed by the dynamic check and action are known. On some other architecture, where register windows are not available, such an approach may be essential for performance. Such partial context switches can also help on the SPARC when there are window spills and refills (i.e., the window is saved or restored from memory).

From Table 1, the cost of instrumentation is very high compared to the number of instructions executed because of the presence of several branch/jump/call instructions (which ranged from 5 in the permanent instrumentation to 9 in the coupled instrumentation). In transient and coupled instrumentation, the flush instruction is used to flush the hardware cache, which can hurt performance. From the figure, it appears that permanent instrumentation is the least expensive instrumentation technique. However, the other techniques can have lower cost, depending on how often the instrumentation needs to be removed.

Hardware cache simulation: The first application where we used our instrumentation approach was a hardware cache simulator [1]. For this purpose, we instrumented the first instruction of each fragment and every load and store instruction. Since our system inserts the instrumentation at fragment creation time, it is possible to make a single call to the instruction cache simulator per fragment with the base address of the fragment in the application binary and the number of instructions in the fragment as arguments. The simulator can simulate the I-cache for each of the instructions in that fragment with this information. For the D-cache, we instrumented each load and store instruction. The static part of the trigger checked whether an instruction is a load or a store for the data cache simulation. The static check always returned true for the instruction cache simulation. For this application, the dynamic part of the trigger made a call to the action to send a memory reference to the cache simulation (for the instruction or data cache). The action routines also computed the effective address for the memory reference.

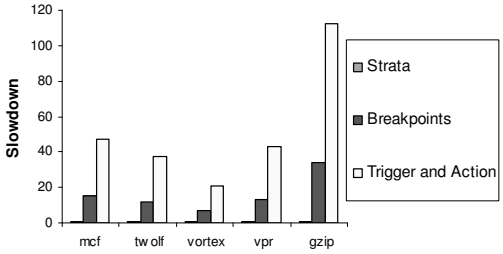

Figure 9: Slowdown in Cache simulation experiments

Figure 9 shows the breakdown of the slowdown for five SPEC2000 benchmarks. The run-time for each of the benchmarks has been normalized to the run-time of the application without any instrumentation. We see that most of the overhead comes from the action (the dynamic check is lightweight in this case). The fast breakpoints account for the next biggest part of the run-time.

Profiling edge counts: For another application, we needed to profile the program to find the edge counts of the fragments in the fragment cache. Any two fragments that execute successively constitute an edge. We wanted to collect such a profile using the SDT system to change the layout of the code in the fragment cache in Pettis-Hansen style [18]. For this purpose, we needed to have one instrumentation point per fragment and keep the instrumentation permanently. We ran the same set of benchmarks as above with and without the instrumentation and measured the cost of the instrumentation.

For this experiment, the static check looked for an instruction that is the first instruction in the fragment. The dynamic check always called the action. The action did a hash lookup for the current edge and inserted a new edge if the hash lookup failed or incremented a count if it succeeded. Figure 10 shows the breakdown of costs incurred in this experiment.

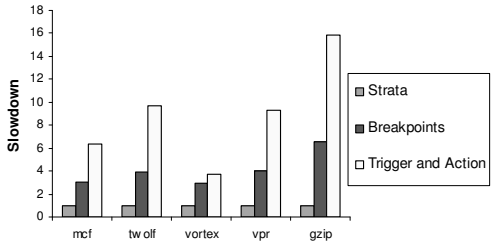

Figure 10: Slowdown in block placement experiments

Again, this figure demonstrates that action accounts for the biggest part of the run-time overhead. In this case, the slowdown of the program is not as high as in cache simulation. This is because cache simulation involved a lot more instrumentation points than this experiment, as every load and store instruction in the program was instrumented in the former case. In this experiment *gzip* had the most overhead because it had more hits of the breakpoints than the other benchmarks.

Collecting the working set: For another application, we needed to profile a program in order to collect the set of fragments that are temporally close during execution. We wanted to find out the optimal size of the fragment cache in Strata for every application. This time, we needed temporal information, which would be a lot of data, so we collected the information via sampling the execution of the program. We sampled 10 thousand fragments for every 10 million fragments that were executed.

In this case, the static check of the trigger involved checking for an instruction being the first instruction in a fragment. The dynamic check of the trigger verified that we are in sampling mode; that is, we have executed at least 10 million fragments without sampling and we do not have more than 10 thousand edges in the current sample. The action saved the ID of the current fragment (in sampling mode) and incremented a counter for the number of fragments executed (in sampling and non-sampling modes). We ran the same set of benchmarks, which are shown in Figure 11.

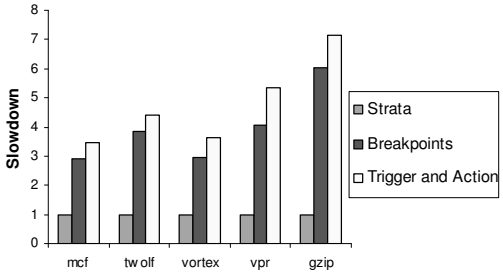

Figure 11: Slowdown in working set experiment

The overhead of the trigger-action system is the lowest for this application. The reason is that the action in this case is very simple. Most of the time, the action involves incrementing a counter. The disparity in the slowdown incurred, once again, depends on how many times the breakpoints are hit. Although the number of hits to the breakpoint in this case and in the previous case (Figure 11) is exactly the same, the trends in the two figures are slightly different. This is due to the fact that the previous experiment involved a hash lookup in the action routines, which would incur a variable amount of cost depending on the number of conflicts in the hash table. This experiment shows that our infrastructure can support different kinds of information gathering including actions whose behavior changes with time.

6. Related Work

Instrumentation techniques have been used in software dynamic translation systems for a number of purposes including dynamic optimizations [3, 4, 5, 6], software security purposes [2, 8], binary translation [7], and code monitoring [9]. In all of the above systems, instrumented code is "hard-coded" into the system. In Dynamo [3], the instrumentation happens in the interpreter and once the code has been emitted to the fragment cache, new instrumentation inside the fragment cache would involve flushing the cache. Walkabout [7] works in a similar manner. In the case of Dynamo RIO [5, 6, 8], instrumentation is typically at the edge of basic blocks; when it is done inside a basic block (while sandboxing a system call), removing the instrumentation requires flushing the fragment cache. The DELI system [9] is similar. In Jalapeno [4], *yield points* are instrumented at method prologues and loop back edges. Adding new yield points in such a system would be difficult and removing the existing ones would also be difficult.

The concept of fast breakpoints [16] was introduced by Kessler. In that work, the author used the technique that

we referred to as *permanent instrumentation*. The fast breakpoints were not applied in a flexible manner and there was no general infrastructure for doing such instrumentation.

The code modification systems Vulcan [10] and Dyninst [11] used a technique similar to our technique in order to instrument a running program. They made use of fast breakpoints to instrument the binary. However, both of these systems are built for very specific purposes and to the best of our knowledge did not have retargetability or reconfigurability in mind. The Vulcan system was designed for distributed systems to do program transformations and optimizations. The dyninst work was meant for performance monitoring of parallel systems. In [12], an instrumentation system was implemented for monitoring, debugging and profiling OS kernels.

7. Summary

In this paper, we presented a flexible instrumentation approach for software dynamic translators. Our approach uses a trigger-action mechanism that applies static property checks during code generation and dynamic property checks during code execution. An associated action can be invoked when a property is satisfied to gather information about the executing program. In the paper, we showed three different mechanisms for instrumentation and compared their memory and performance costs. We also showed three applications of information gathering to demonstrate the flexibility of our approach to supporting different instrumentation needs.

References

[1] K. SCOTT, N. KUMAR, S. VELUSWAMY, B. CHILDERS, J. DAVIDSON, M.L. SOFFA. *Reconfigurable and Retargetable Software Dynamic Translation*, In Proceedings of the First Annual IEEE/ACM International Symposium on Code Generation and Optimization. San Francisco, California, March 2003.

[2] K. SCOTT, AND J. DAVIDSON. *Safe Virtual Execution Using Software Dynamic Translation*, In Proceedings of the 2002 Annual Computer Security Application Conference, Las Vegas, Nevada, December 9-13, 2002.

[3] BALA, VASANTH, E. DUESTERWALD, AND S. BANERJIA. *Dynamo: A Transparent Dynamic Optimization System*. Proc. of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation, 2000, pp. 1-12

[4] M. ARNOLD, S. FINK, D. GROVE, M. HIND, AND P. SWEENEY. *Adaptive optimization in the Jalapeno JVM.* In Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '00), pages 47--65, Oct. 2000.

[5] D. BRUENING, E. DUESTERWALD, AND S. AMARASINGHE. *Design and implementation of a dynamic optimization framework for Windows*. In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4), December 2000.

[6] D. BRUENING, T. GARNETT AND S. AMARASINGHE. *An Infrastructure for Adaptive Dynamic Optimization.* In Proceedings of the First Annuak IEEE/ACM International Symposium on Code Generation and Optimization, San Francisco, California, pages 265-275, March 2003

[7] C. CIFUENTES, B. LEWIS, AND D. UNG. *Walkabout - a retargetable dynamic binary translation framework.* Technical Report TR2002 -106, Sun Microsytems Laboratories, Palo Alto, CA 94303, January 2002.

[8] V. KIRIANSKY, D. BRUENING, AND S. AMARASINGHE. *Secure Execution Via Program Shepherding.* In 11th USENIX Security Symposium, August 2002.

[9] G. DESOLI, N. MATEEV, E. DUESTERWALD, P. FARABOSCHI, AND J. FISHER, *DELI: A New Runtime Control Point*. Proc. of MICRO-35, Nov. 2002.

[10] A. SRIVASTAVA AND A. EDWARDS. *Vulcan: Binary Transformation in a Distributed Environment.* Microsoft Research Tech. Rpt. MSR-TR

[11] J. HOLLINGSWORTH, B. MILLER AND J. CARGILLE. *Dynamic Program Instrumentation for Scalable Performance Tools. SHPCC*, Knoxville Tennessee, May 1994.

[12] A. TAMCHES AND B. MILLER. *Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels.* Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99), pp. 117-130. New Orleans, LA, February 1999. USENIX.

[13] R. CMELIK AND D. KEPPEL. *Shade: A fast instruction-set simulator for execution profiling.* Technical Report 93-06-06, Department of Computer Science and Engineering, University of Washington, June 1993.

[14] E. WITCHEL AND M. ROSENBLUM. *Embra: fast and flexible machine simulation*. In Proceedings of the 1996 SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Philadelphia, May, 1996.

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

[16] P. KESSLER. *Fast Breakpoints: Design and Implementation*. In Proceedings ACM

SIGPLAN'90 Conf. on Programming Languages Design and Implementation, pages 78-84, 1990.

[17] K. SCOTT AND J. DAVIDSON. *Strata: A Software Dynamic Translation Infrastructure*, In Proceedings of the IEEE 2001 Workshop on Binary Translation, Barcelona, Spain, September 8, 2001

[18] K. PETTIS AND R. HANSEN. *Profile guided code positioning*. Proc. ACM SIGPLAN'99 Conf. on Programming Languages Design and Implementation, pages 16-27, June 1990.

[19] C. JARAMILLO, R. GUPTA, M. L. SOFFA. *FULLDOC: A Full Reporting Debugger for Optimized Code*. In Proceedings of Static Analysis Symposium, 2000