

Retargetable and Reconfigurable Software Dynamic Translation

K. Scott*, N. Kumar†, S. Velusamy*, B. Childers†, J. W. Davidson*, and M. L. Soffa†

**Department of Computer Science
University of Virginia
{kscott,siva,jwd}@cs.virginia.edu*

*†Department of Computer Science
University of Pittsburgh
{naveen,childers,soffa}@cs.pitt.edu*

Abstract

Software dynamic translation (SDT) is a technology that permits the modification of an executing program's instructions. In recent years, SDT has received increased attention, from both industry and academia, as a feasible and effective approach to solving a variety of significant problems. Despite this increased attention, the task of initiating a new project in software dynamic translation remains a difficult one. To address this concern, and in particular, to promote the adoption of SDT technology into an even wider range of applications, we have implemented Strata, a cross-platform infrastructure for building software dynamic translators. This paper describes Strata's architecture, our experience retargeting it to three different processors, and our use of Strata to build two novel SDT systems—one for safe execution of untrusted binaries and one for fast prototyping of architectural simulators.

1. Introduction

Over the years software dynamic translation (SDT)—the programmatic modification of a running program's binary instructions—has become an increasingly useful technique in the system implementor's repertoire. A wide variety of systems can be classified as software dynamic translators. Among these are debuggers, dynamic optimizers, dynamic binary translators, dynamic instrumentation systems, dynamic software updaters, and certain emulators and simulators. Despite apparent differences, these systems are fundamentally similar: they virtualize aspects of the host execution environment by interposing a layer of software between program and CPU. This software layer acts as a virtual machine that mediates program execution by dynamically examining and translating some or all of a program's instructions before they are run on the host CPU.

Software dynamic translation affords system designers unprecedented flexibility in controlling and modifying a program's execution. This flexibility allows software dynamic translation to be used to accomplish a

variety of objectives not easily achieved via other means. For instance, software dynamic translation may be used to overcome the barriers to entry associated with the introduction of a new OS or CPU architecture. Transmeta's Code Morphing technology is used for this very purpose, i.e., allowing unmodified Intel IA-32 binaries to run on the low-power, VLIW Crusoe processor [7]. Similarly, the UQDBT system dynamically translates Intel IA-32 binaries to run on SPARC-based processors [23], and FX!32 dynamically translates x86 binaries to run on Alpha processors [4].

In addition to allowing designers to overcome cost barriers to new platform acceptance, the flexibility of software dynamic translation has proven useful for a variety of other purposes. For instance, Shade uses software dynamic translation to implement high-performance instruction set simulators [6]. Embra uses software dynamic translation to implement a high-performance operating system emulator [26]. Dynamo and Mojo [1, 3] use software dynamic translation to improve the performance of native binaries, and Daisy uses software dynamic translation to evaluate the performance of novel VLIW architectures and accompanying optimization techniques [8]. The Kerninst [21, 20] and Vulcan [19] systems use software dynamic translation to insert performance monitoring instrumentation into running programs. And most recently software dynamic translation has been used to ensure safe execution of untrusted binaries [14, 17, 18].

Despite the many uses to which SDT has been put and the lively state of research into novel uses of SDT, it still remains a significant challenge to build SDT systems. Software dynamic translators are typically written for a single application and/or platform. For example, the Shade simulator was written specifically to simulate MIPS, SPARC v8, and SPARC v9 programs on SPARC v8 architectures [6]. Because Shade was not written with other architectures or other potential uses in mind, it would be a non-trivial task to retarget Shade to new architectures or to add new functionality, e.g., optimization. The single-target, single-purpose approach to software dynamic translator design places a significant

burden on designers of new SDT systems. Developing a new system from scratch with no opportunity for software reuse imposes a steep learning curve and inevitably requires a substantial software development effort.

To address the difficulty of building software dynamic translators and to promote research into novel uses of SDT, we have developed an SDT implementation infrastructure called Strata. Strata provides a common framework in which researchers can build software dynamic translators. This common framework promotes code reuse in two ways. First, Strata provides simple software dynamic translators for a variety of architectures; researchers can modify these translators to suit their specific needs without having to build an entire translator from scratch. Second, the standard implementation framework provided by Strata allows code reuse through composition; researchers can use the work of others to enhance their dynamic translators. For example, by composing a Strata-based dynamic optimizer with a Strata-based simulator, an optimizing simulator can be realized.

In this paper we present the design of the Strata retargetable software dynamic translation framework. We illustrate the ease with which Strata-based dynamic translators can be developed by presenting two significant and useful software dynamic translators: a system-call monitor and a cache simulator.

The remainder of this paper has the following organization. Section 2 details the design of Strata and our experience retargeting it to three popular platforms. Section 3 describes a novel system call monitor based on SDT and implemented using Strata. Section 4 describes a high performance, SDT-based cache simulator implemented with Strata. Finally, section 6 summarizes our findings.

2. The Design of Strata

2.1. Overview

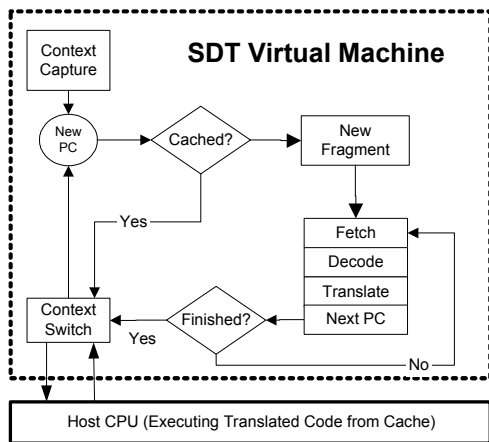
In the Fall of 2000 we began a joint project between the Zephyr Research Group at the University of Virginia and the Compiler and Architecture Group at the University of Pittsburgh to build a retargetable dynamic optimizer. As we surveyed the dynamic optimization literature we noticed that several of the techniques employed by Dynamo and Mojo [1, 3] had been previously employed in simulation work by Cmelik and Keppel [6] and in emulation work done by Witchel and Rosenblum [26]. This reuse of techniques amongst seemingly disparate applications—PA-RISC and x86 hosted dynamic optimizers, a SPARC hosted simulator, and a MIPS-hosted operating system emulator—

inspired us to cast our net wider. As we began to review the broader literature in dynamic optimization, dynamic binary translation, emulation, simulation, performance monitoring, and late program modification we observed two things: many systems dynamically modify binary instruction streams in order to accomplish some specific functionality, be it enhancing the performance of native binaries, executing non-native binaries, or something entirely different; and all of these systems share a common set of features, e.g., caching and linking blocks of translated code.

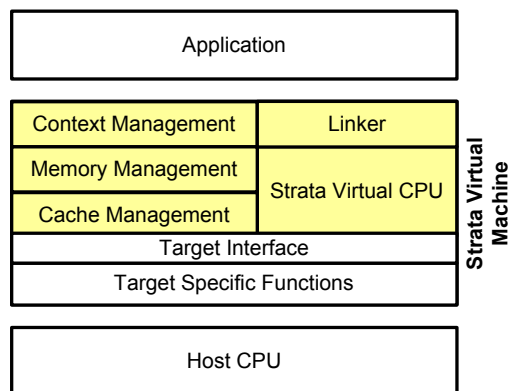
Given our interest in retargetable systems software and compiler development frameworks we set out to build an infrastructure that would promote development and reuse of software dynamic translators¹ through the provision of a set of simple translators that are easy to reconfigure and retarget. To accomplish both goals—reconfigurability and retargetability—we organized Strata around the conceptual model of a virtual machine (see Figure 1). The Strata VM mediates application execution by examining and translating instructions before they execute on the host CPU. Translated instructions are held in a Strata-managed cache. The Strata VM is entered by capturing and saving the application context (e.g., PC, condition codes, registers, etc.). Following context capture, the VM processes the next application instruction. If a translation for this instruction has been cached, a *context switch* restores the application context and begins executing cached translated instructions on the host CPU.

If there is no cached translation for the next application instruction, the Strata VM allocates storage for a new *fragment* of translated instructions. A fragment is a sequence of code in which branches may appear only at the end. The Strata VM then populates the fragment by fetching, decoding, and translating application instructions one-by-one until an end-of-fragment condition is met. The end-of-fragment condition is dependent on the particular software dynamic translator being implemented. For many translators, the end-of-fragment condition is met when an application branch instruction is encountered. Other translators may form fragments that emulate only a single application instruction. In any

1. We use the term “software dynamic translator” to generically refer to any system that dynamically modifies an executing instruction stream through software. The software qualifier is significant, given that most modern CPU’s do some form of dynamic translation in hardware, e.g., out-of-order execution [10]. The term dynamic binary translator is also inappropriate for labeling this entire class of software tools, given that most researchers use dynamic binary translator to mean a piece of software that dynamically translates a binary from one instruction-set architecture to another, e.g., x86 to Alpha or PowerPC to VLIW [4, 8].



(a)



(b)

Figure 1: Strata Architecture

case, when the end-of-fragment condition is met, a *context switch* restores the application context and the newly translated fragment is executed.

As the application executes under Strata control, more and more of the application's working set of instructions materialize in the fragment cache. This, along with certain other techniques that reduce the number and cost of context switches, permits Strata to execute applications with little or no measurable overhead [16].

Figure 1(b) shows the components of the Strata VM. The Strata virtual machine is implemented as a set of target-independent *common services*, a set of *target-specific functions*, and a reconfigurable *target interface* through which the machine-independent and machine-dependent components communicate. Implementing a new software dynamic translator often requires only a small amount of coding and a simple reconfiguration of the target interface. Even when the implementation is more involved, e.g., when retargeting the VM to a new platform, the programmer is only obligated to implement the target-specific functions required by the target interface; common services should never have to be reimplemented or modified. The target interface and common services are beyond the scope of this paper and are detailed elsewhere [16, 17].

2.2. Strata-SPARC

The first Strata software dynamic translator was developed for the SPARC V8/V9 instruction set architecture [25] and the Solaris operating system. Predictably, the Strata-SPARC target was relatively simple to implement with the few minor complications detailed below.

SPARC instructions are uniform in length and have a relatively simple encoding. Consequently, the Strata-SPARC VM decode and fetch routines were very easy to implement. Furthermore, most SPARC instructions require only trivial manipulation during translation. The exceptions to this rule are the SPARC's delayed control transfer instructions. A delay slot instruction (DSI) on the SPARC can be associated with any control transfer instruction, and is issued into the pipeline before the control transfer instruction. On the SPARC, DSIs may also be annulled, i.e., not executed if the branch is taken or not-taken.

The complex semantics of the SPARC's delayed control transfers require careful attention when performing certain optimizations. For instance, partial inlining of function calls requires the elimination of a call instruction. When such a control transfer instruction is to be eliminated, the side effects of the control transfer in addition to the effects of the DSI must be emulated. On the SPARC a call transfers control to a signed 32-bit offset from the call instruction's location in the program text. The call also writes the value of the PC (address of the call instruction) into register %o7, and executes the DSI before the control transfer takes place. The value of the PC, placed into %o7 is visible to the DSI, even though the control transfer is delayed. Consequently, when eliminating a call for the purpose of partial inlining, Strata must emit instructions to load a PC value into %o7 and to perform the effects of the DSI.

The other significant point of interest associated with the Strata-SPARC target is the implementation of the context switches that transfer control between the Strata VM and the application. In general, the time spent executing these context switches is the single

largest factor in SDT performance. A variety of target-independent techniques can be used to reduce the number of context switches, including fragment linking, partial inlining of function calls, and indirect branch elimination [16]. Even though highly effective, not all context switches can be avoided using these techniques. Consequently, it is prudent to implement context switches in the most efficient manner possible in order to reduce the time spent executing them.

To implement a context switch on the SPARC, we must save or restore the general-purpose registers, the condition codes, and several status registers. SPARC register windows facilitate this process by allowing groups of general-purpose registers to be saved and restored with a single instruction. Using register windows, the context switch code for the SPARC is extremely compact, requiring only 22 instructions.

Other potential SDT implementation difficulties were avoided due to several serendipitous design choices made by the SPARC's architects. For instance, the PC on the SPARC is a special register that can only be read using a special instruction. When Strata-SPARC encounters this instruction it emits code to load the original text segment PC value into the given registers. This allows self-inspecting programs, e.g., applications that compute a checksum over a portion of their text segment, to run seamlessly under Strata-SPARC. The SPARC architecture also requires modifications to the program text segment to be explicitly synchronized with the instruction cache through a flush instruction. Strata-SPARC forces either a full or partial fragment cache flush when SPARC flush instructions are encoun-

tered. This relatively simple mechanism allows Strata-SPARC to correctly execute programs that include self-modifying code.

2.2.1. Experiments

Using SPEC 2000, we compared the performance of the SPEC benchmarks running under Strata-SPARC with native execution on the host machine (a 360MHz UltraSparc II with Solaris 7 and 1G of RAM). All programs are compiled with Sun CC version 5.0 with the flags `"-fast -xO5 -arch=v8plus"`. As Figure 2 shows, the applications in the SPECint2000 benchmark suite run with small slowdowns that range from 1.02x to 1.8x with an average slowdown across all benchmarks of 1.32x.

2.3. Strata-MIPS

Our second target for Strata was the MIPS IV instruction set architecture and the IRIX 6.5.10 operating system. This new target, Strata-MIPS, was based heavily on the SPARC target. In porting Strata to the MIPS, we found Strata's structure to be flexible and relatively easy to retarget. The initial version of Strata-MIPS took one person less than three weeks of development effort due to Strata's highly modular structure. The main challenges that we encountered involved context switches and instruction set differences between the MIPS and the SPARC. We discuss each of these challenges below.

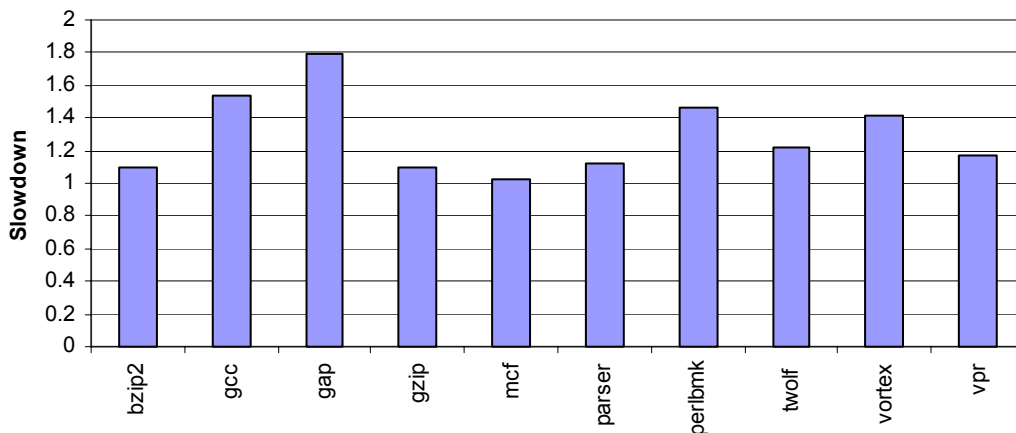


Figure 2: Performance of Strata-SPARC

The major effort in retargeting Strata-SPARC to MIPS involved writing a new instruction decoder. The MIPS has a regular architecture that made it relatively simple to implement a very fast instruction decoder. The modular structure of Strata, particularly the decode and translate steps, let us quickly include a prototype decoder and bootstrap the system within a few days of starting the development effort.

The MIPS' architecture is similar to the SPARC V8, with a few minor differences. One such difference is the inclusion of a branch-likely instruction in MIPS IV. Although not necessary for correctness of the translator, we tried to use the MIPS' branch-likely instruction as a hint to prefetch the target block of a branch-likely and to link the prefetched fragment with the branch-likely fragment. To add support for branch-likely required only small changes in the decoder and the translator (to prefetch a fragment). Unfortunately, prefetch support had a small performance gain (at most 1%).

Unlike the SPARC, the MIPS instruction set architecture does not include explicit instruction cache flush instructions; instead, instruction cache flushes are performed using an OS call. Although we could intercept the OS call to flush the fragment cache for self-modifying code, we did not encounter any such flushes in our benchmarks.

A final difference between MIPS and SPARC is that MIPS dynamically linked libraries (DLLs) are fully linked at binary load time, before the Strata VM is first invoked. Hence, Strata-MIPS did not have to handle the linkage mechanisms and cache flushes normally associated with loading DLLs.

As with Strata-SPARC, the Strata-MIPS target must perform uneliminated context switches as efficiently as possible in order to minimize SDT overhead. The context switch mechanism in Strata-MIPS is similar to the mechanism in Strata-SPARC. One difference is that the MIPS does not have instructions that save and restore an entire set of registers. To do a context switch in Strata-MIPS requires that a sequence of load and store instructions be executed to save and restore the application's general-purpose registers to and from the stack. This context switch takes 78 or 84 instructions (the instruction count depends on whether a fragment terminates with a branch or a jump), while it takes only 22 instructions in Strata-SPARC. The relatively high overhead of a full context switch in Strata-MIPS increases the importance of minimizing the number of control transfers between the application and the builder.

Strata-MIPS context switches involve two special cases not encountered in Strata-SPARC. As is the case with Strata-SPARC, Strata-MIPS uses an indirect jump to transfer control between the application and the Strata VM. When transferring control to the VM, the

context of the application is saved and the fragment target address is extracted and passed as a parameter to the VM. However, when the VM returns control to the application, it must restore the application context and execute an instruction to transfer control to the next fragment. The control transfer from the VM to the fragment is done with an indirect jump through a register. The jump itself should be executed after the context of the application is restored. Restoring the full context overwrites the register that holds the target address for the context switch. Our solution is simple: The application context is partially restored, except for the register holding the target address (\$31). A load at the top of every fragment restores \$31 once control has been transferred to the fragment. When linking fragments, we ensure that the fragment link is directed to the second instruction in a fragment (the restore of \$31 is only required for a context switch).

The second special case for Strata-MIPS context switches arises when a fragment is terminated by an indirect jump. When this occurs, the instruction in the delay slot of the jump may use the same register as the jump's target register. Because the DSI is always executed, the context of the application includes the value of the destination register computed by the DSI. This case is handled by extracting and saving the contents of the indirect jump's register (either in a register or on the stack) before executing the DSI. The DSI's destination register is then saved to the application context, and the target address register is restored to do the control transfer.

2.3.1. Experiments

Using SPEC 2000, we compared the performance of the SPEC benchmarks with Strata-MIPS and the performance of running them natively on the host machine (a dual-processor SGI platform running IRIX 6.5.10 with 250 MHz MIPS R10000 processors and 1GB of memory).

Figure 3 shows the run-time performance of Strata normalized to the host machine running the benchmarks natively. For the MIPS R10000, we were able to run seven of the SPEC 2000 benchmarks. Three of them did not run natively on the machine (without Strata) and we did not consider these in the results. For most of the benchmarks, Strata-MIPS incurred minimal slowdown with a range of 1.09x to 3.0x and an average across all benchmarks of 1.8x. Two of the benchmarks, *perl* and *gcc*, had high overheads of 2.6x and 3.0x. For both benchmarks, the high overheads result from Strata-MIPS re-entering the fragment builder a large number of times. For example, in *perl*, there were over 5.3 million builder re-enters, mostly due to indirect

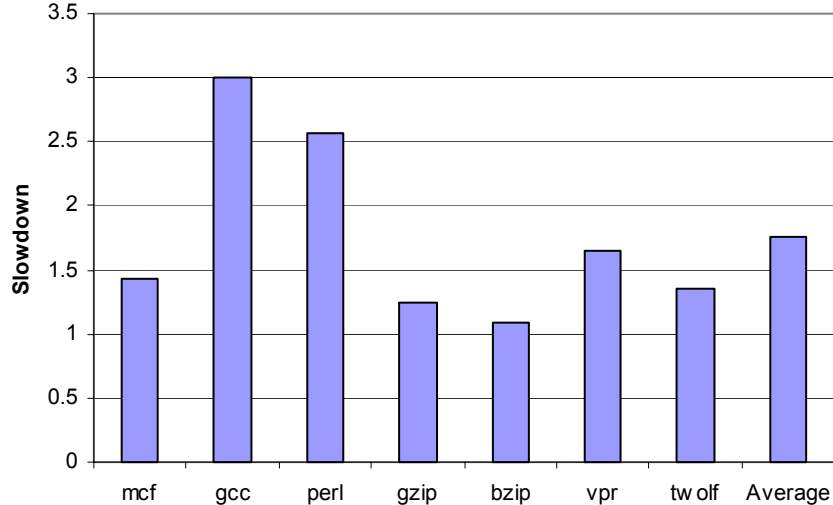


Figure 3: Performance of Strata-MIPS

branches which could not be handled by one of our indirect branch elimination techniques [16]. As we have remarked in previous sections, these context switches, due to either indirect branches or a large number of unique fragments, generally result in high SDT overhead.

2.4. Strata-X86

The last Strata target discussed in this paper is the port to the Intel 80x86 architecture. The x86’s CISC design provided us with a different set of challenges than those encountered during the development of the two RISC targets. Despite the dramatic difference between the x86 and SPARC instruction set architectures, we still used Strata-SPARC as the basis for retargeting to the x86.

Compared with Strata-SPARC and Strata-MIPS, a much larger percentage of overall development time was spent implementing the Strata-X86 instruction fetch and decode functions. Unlike the SPARC and MIPS ports, the Strata-X86 fetch routine has to do some amount of decoding of an instruction to determine its length. Our implementation uses a static table indexed by opcode to get the length of an instruction. For the majority of opcodes, this table lookup is sufficient; however, some instructions require additional decoding to locate their Mod-Reg-RM and SIB bytes so that their lengths can be correctly computed [11].

Aside from fetch and decode, many aspects of the x86 target are simpler to handle than either SPARC or MIPS. For example, the x86 allows operations on memory locations. This capability reduces the need to manipulate registers to compute memory addresses, which decreases context switch overhead. Also, on the

x86, the absence of delayed branches simplified the handling of branch instructions.

Like Strata-SPARC and Strata-MIPS, Strata-X86 needs to save and restore application context, including the general-purpose registers, the flags register, and the stack pointer. Using the aforementioned ability of x86 instructions to operate directly on memory locations the code required to save or restore the x86 application context is small—10 instructions for saving the context and 9 for restoring the context. This is half the size of the Strata-SPARC context switch code and an order of magnitude smaller than the code required to switch contexts under Strata-MIPS.

2.4.1. Experiments

To measure the overhead of Strata, we compare the performance of the SPEC 2000 benchmarks running under Strata and running natively. The benchmarks were compiled with gcc with -O3 -fomit-frame-pointer and were run on an unloaded dual processor Athlon 1800+ running at 1.6GHz with 1 GB of RAM.

Figure 4 shows the execution times relative to native hardware execution. The first set of bars shows Strata-x86’s performance on the SPEC 2000 benchmarks. The run-time overhead varied from approximately 1.0 to 1.8, with an average overhead of 1.35. The second set of bars in the figure show the overhead of another x86 SDT system, called DynamoRIO [2, 14]. The third set of bars show DynamoRIO’s overhead without trace formation. The results show that Strata-x86 performs comparable with DynamoRIO (which is engineered specifically for the x86 architecture) for half the benchmarks and is about 10% slower for the rest. We attribute

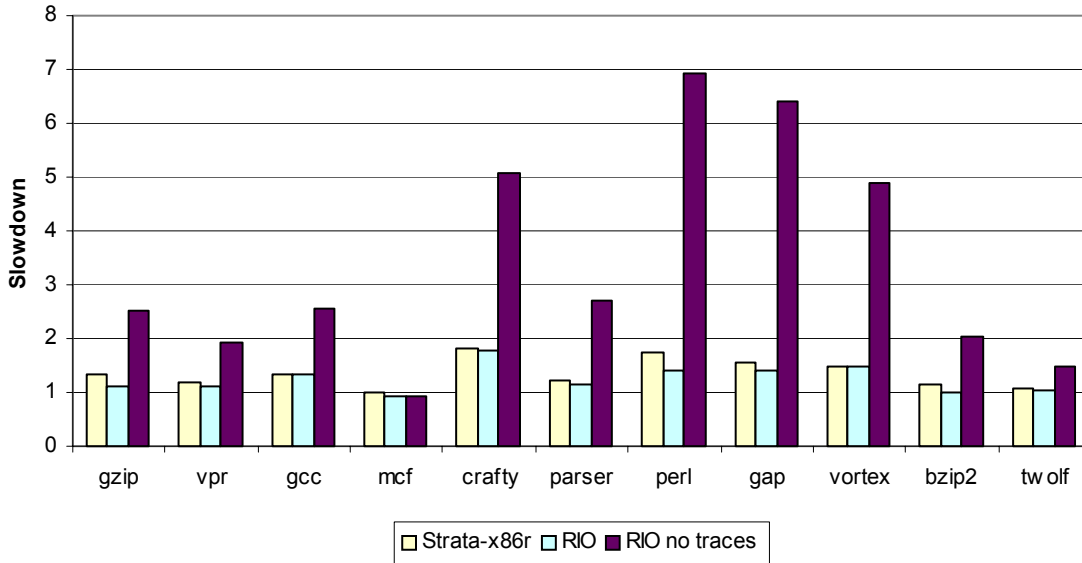


Figure 4: Performance of Strata-x86

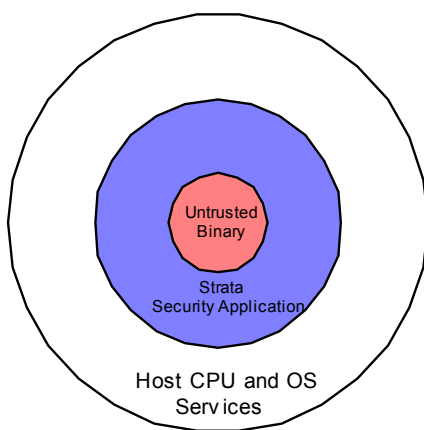
this difference to the trace construction employed by DynamoRIO.

3. System call monitoring with SDT

SDT’s ability to control and dynamically modify a running program provides an ideal mechanism for implementing a system call monitor. A system call monitor observes invocations of operating system calls and determines whether and how to let those system calls execute. System call monitors thereby explicitly control an application’s access to system resources, and can be used to enforce a variety of security policies to improve the safety and reliability of untrusted binary code.

Strata provides an ideal platform for implementing a system call monitor [18]. As the untrusted binary is virtualized and executed by Strata, code is dynamically inserted to intercept system calls and to potentially redirect those calls to user supplied functions. In general, this process does not need to be limited to system calls. All access to host CPU and operating system resources are explicitly controlled by Strata (see Figure 5).

A simple, but realistic example illustrates our approach. Suppose a user wishes to enforce a policy that prohibits untrusted applications from reading a file that the user normally has permission to read. Let’s call this file `/etc/passwd` (`registry.dat`, `SAM`, or `system` might be equally good choices). Now assume the



```
int myopen (const char *path, int oflag) {
    char fname[FNAMESIZE];
    int fd;

    strata_policy_begin(SYS_open);
    makepath_absolute(fname,path,FNAMESIZE);
    if (strcmp(fname,"/etc/passwd")==0) {
        strata_fatal("Naughty, naughty!");
    }
    fd = syscall(SYS_open,path,oflag);
    strata_policy_end(SYS_open);
    return fd;
}
```

Figure 5: Strata Resource Control

user receives an untrusted binary called `funny` and wishes to run it. The user runs `funny` using Strata. The Strata loader locates the entry point of the application and inserts a call to the Strata startup routine. When the loader begins execution of the application, the call to the Strata startup routine causes Strata to be dynamically loaded and invoked.

As Strata processes `funny`'s text segment and builds fragments to be executed, it locates open system calls and replaces them with code that invokes the system call policy code. When the fragment code is executed, all open system calls are diverted to the policy code (`myopen()` in Figure 5). It is the policy code's job to examine the arguments to the original open system call. If the untrusted application is attempting to open `/etc/passwd`, an error message is issued and execution of the application is terminated. If the file being opened is not `/etc/passwd`, the security policy code performs the open request and returns the result and execution continues normally (albeit under the control of Strata).

In order to implement the system call monitor described above, Strata's default translate function that processes trap or interrupt instructions must be overridden. On the SPARC/Solaris platform less than 20 lines of new code is required to implement the new translate function.

For each operating system call site, the overridden translate function tries to determine if the operating system call is one to be monitored. In most cases, Strata can determine at translation time the operating system call to be invoked. If the OS call is one to be monitored, the code to invoke the operating system call is replaced with a call to the user-supplied code. If the call is not one to be monitored, no translation action need be taken and the operating system call code is copied unchanged to the fragment cache.

In some cases, the operating system call to be invoked cannot be determined at translation time. This situation can occur with indirect operating system calls. In this case, Strata must generate and insert code that, when the fragment is executed, will test whether the OS call being invoked is one to be monitored. If so it must call the appropriate user-supplied policy code; otherwise, the OS call is executed.

In addition to the ease with which system call monitors can be built with SDT, there are three significant advantages to using SDT for mediate access to system services. First, system call monitors built in Strata are user-level software; unlike other system call monitors, such as Janus [9] and interposition agents [12], Strata requires no special operating system services. Second, because Strata inlines policy code into the executing application, Strata can perform monitoring without

expensive additional context switches and/or function calls. Third, unlike systems which statically instrument binaries to perform safety checks, Strata can handle self-modifying code, e.g., malicious code generated by local buffer overrun exploits.

4. Fast cache simulation using SDT

Another example of the Strata's usefulness for implementing a variety of differing applications can be demonstrated through the implementation of a high-performance architectural simulator. In a separate research project, we had a need for a fast cache simulator that works in concert with online application profiling and dynamic optimization. This need motivated our desire to build a cache simulator and a general framework for architecture study using Strata. We found that SDT has several advantages for processor simulation. One advantage is that it is easy to manipulate the dynamic execution of a program to model different architecturally visible features. This flexibility makes it possible to quickly and virtually introduce new features into a processor to experiment with the software/hardware interface and to observe the impact of the new changes (e.g., how can a code optimizer take advantage of new instructions for value prediction).

Another advantage to building simulators with Strata is that they are portable across different machine architectures, such as the three architectures discussed in this paper. A final advantage is that simulating new processor features with SDT is very fast because the application binary runs natively on the underlying machine. Such a flexible and fast simulation mechanism is desirable since it can be used to quickly prototype a new feature and pose "what if" questions. Lightweight simulators built with Strata can be used as part of the design cycle during initial evaluation of a feature and full execution-driven simulation used when the initial measurements look encouraging.

Strata can dynamically insert code into a program to model an architecture change and to gather statistics about the program with the new feature. Code instrumentation has good performance relative to execution-driven simulation because the host machine directly executes the binary program. The downside is that it is difficult to model certain changes to the microarchitecture, particularly ones that are not architecturally visible (e.g., changing the number of pipeline stages). In our work, instead of statically instrumenting the binary program, we use Strata to instrument the program as it executes. Thus, only executed paths are instrumented. Furthermore, the instrumentation can be inserted and removed based on run-time characteristics, which is difficult with a purely static approach.

The basis of our framework uses a trigger-action mechanism to intercept the execution of a program to model a processor feature. Triggers check code properties and actions perform some function when a property holds. A trigger is installed during code fragment generation and can be associated with any instruction in a fragment. The attachment of triggers to instructions involves checking a static property. In this case, a static property is some aspect of the current code fragment that can be checked as the fragment is being generated. For example, checking the opcode of an instruction to see whether it is a load or store is a static check. During fragment execution, when a trigger is encountered, an optional dynamic property can be checked. This dynamic check may inspect any part of the machine state, including the values of registers, memory locations, program counter address, and fields within an instruction. When a property check holds, an associated action is performed. In our cache simulator, a static check is done on every instruction installed in the fragment cache. That static check identifies which instructions involve a memory reference (i.e., a load or store). If an instruction is a load or store, a trigger is attached to that instruction. When the trigger is encountered during a fragment's execution, an action is fired that performs cache simulation with the memory reference. Likewise, the trigger-action mechanism can emulate newly introduced instructions by putting a trigger on the new instruction that fires an action that does the semantics for the new instruction.

Trigger-action pairs are implemented with fast breakpoints [13] that are placed at instructions that satisfy a static property. A fast breakpoint replaces an instruction with a jump to a breakpoint handler that does whatever semantic action is required. The handler also executes the replaced instruction. During code execution, when a fast breakpoint is hit, control is transferred to a separate piece of code that checks a dynamic property. If the dynamic property holds, then control transfers to the action. One advantage of fast breakpoints is they are non-intrusive and can be easily removed and inserted dynamically. Insertion replaces the original instruction with a jump and deletion replaces the jump with the original instruction.

In the data cache (D-cache) simulator, a fast breakpoint is installed at every load and store to capture the memory reference for the data cache. When a fast breakpoint corresponding to a memory reference is executed, control transfers immediately to the action for that breakpoint (there is no dynamic check), which updates the cache simulation. An action for a load or store extracts from the machine state the information needed to compute the effective address for that instruction. With the effective address and reference informa-

tion, the D-cache simulator is invoked. Because we only instrument loads and stores from the original binary and they are executed in original program order, the memory stream presented to the cache simulator is the same one as that presented to the underlying hardware when the program is run without SDT.

One issue with using fast breakpoints for the trigger linkage mechanism is a breakpoint may need to be placed on an instruction in the delay slot of a branch. However, most machines with delayed branches do not allow filling the delay slot with another branch (to insert the breakpoint). Our approach installs the breakpoint at the branch itself, rather than on the instruction in the delay slot. For our cache simulator, the application context (the set of general-purpose registers) at the branch instruction is the same as it is at a load or store in the delay slot. Hence, we can safely compute the effective address for the memory instruction at the branch.

Our approach can also model an instruction cache (I-cache). In this case, a trigger is attached at the first instruction in a fragment to invoke the I-cache simulator. A single call is made per fragment to the simulator to minimize the amount of instrumentation executed. During fragment generation, the size of the fragment is known, and a breakpoint can be placed that calls the I-cache simulator with the fragment's base address and number of instruction references. However, the base address is the address in the original binary program, rather than the address of the instructions in the fragment cache. This ensures that the memory address stream seen by the cache simulator is the same as that would be seen during code execution without SDT. One complication is that SDT may insert instructions in the fragment (e.g., trampoline code) or may apply local optimizations (e.g., Strata inlines the first block of a called procedure). In both cases, the information sent to the cache simulator can be "fixed up" to ensure that it matches the original binary.

We implemented a cache simulator in Strata-SPARC that models the first level of a memory hierarchy. The modeled cache has a split I and D organization, where both caches are 16 KB, 2-way set associative with LRU replacement, and have a 32-byte line size. We can model more complex memory hierarchies, but for investigating the overheads and feasibility of using SDT for architectural study, we modeled only one level of the hierarchy. In modeling the cache, there were two issues that we were concerned about. First, we wanted to verify that Strata accurately has the same memory reference stream as the program executing natively without Strata. Second, we wanted to ensure that our system was fast enough to be used for quick studies.

To address the first issue, we compared the memory reference stream from the Shade cache simulator for the SPARC from Sun Microsystems [6] with the memory reference stream from our simulator. Although the results are not reported here, the instruction and data streams from our tool matched the memory reference streams from Shade on the SPEC 2000 benchmarks.

For the second issue, we compared the run-time of our tool to Shade’s run-time. Because Shade is considered state-of-the-art and uses technology similar to SDT, it is an aggressive basis for comparison. In our experiments, the cache simulation code itself very closely models Shade’s simulator to ensure that the comparison is fair by factoring out any performance effects due to different implementations of the simulator itself. Figure 6(a) shows the speedup of Strata-SPARC for cache simulation relative to Shade on several SPEC benchmarks. The Strata-SPARC simulator has a relative speedup over Shade that varies from 1.7 to 2.2, with an average of 1.9. This improvement is due to our tool decoding and instrumenting instructions only once during fragment generation and processing an entire fragment of instruction accesses with a breakpoint.

To get a better understanding of our system’s overhead, we measured the percentage of time spent in running Strata’s fragment builder, the percentage of time spent checking properties and executing fast breakpoints, and the percentage of time spent in cache simulation. Figure 6(b) shows the percentage of run-time associated with different aspects of cache simulation with Strata. As the figure shows, the smallest percentage of time (usually less than 5%) is spent executing the application with Strata, and the largest percentage (about 65% on average) is spent in the cache simulator. Breakpoints can also add a significant amount of time (an average of 30%). One reason for this high overhead

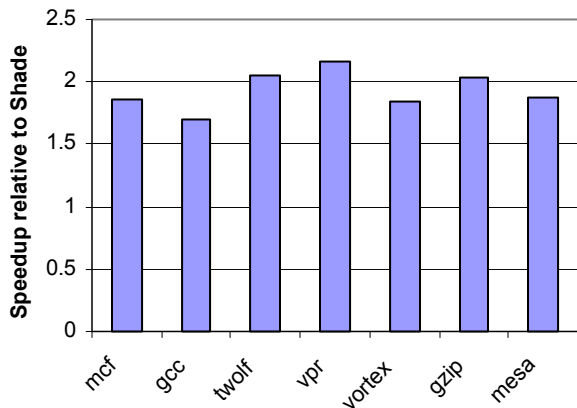
is that our system does a full context switch to invoke the cache simulator. We could do a partial context switch and save only those registers needed by the simulator, but this approach would make it difficult to add new simulation features.

From our experiments with cache simulation in Strata-SPARC, we believe that SDT is well suited for lightweight architecture and compiler experiments. Indeed, we can easily modify our tool to model other architecture features, such as the SPARC’s register windows (to vary the number of available windows and investigate the number of overflows/underflows), different branch predictors to model their accuracy, and emulate new instruction set features (e.g., short vector instructions). We are currently developing a flexible interface and specification language that will allow an user to define their own architectural simulators without having any knowledge of how SDT or Strata works.

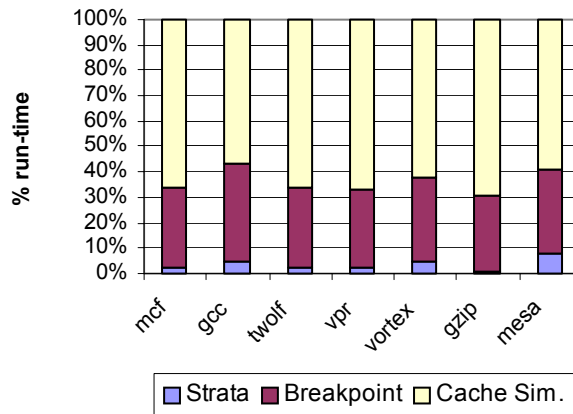
5. Related Work

Software dynamic translation has been used for a number of purposes (see Section 1), including dynamic binary translation of one machine instruction set to another [7, 23, 4, 8], emulation of operating systems (e.g., VMWare, Plex86), and machine simulation [6, 26]. While most of these systems have been built for a single purpose and target to a single machine architecture, there has been some recent work on general infrastructures for SDT.

Walkabout is a retargetable binary translation framework that uses a machine dependent intermediate representation to translate and execute binary code from a source machine on a host machine [5]. It analyzes the code of the source machine to determine how to translate it to the host machine or to emulate it on the host.



(a) Speedup with Strata-SPARC over Shade



(b) Run-time overhead for cache simulation

Figure 6: Cache simulation with Strata-SPARC

Walkabout uses machine specifications to describe the syntax and semantics of source and host machine instructions and how to select hot paths. The current implementation supports only binary translation and has been ported to the SPARC and the x86 architectures. Yirr-Ma is an improved binary translator built with the Walkabout infrastructure [22].

Another flexible framework for investigating SDT is DynamoRIO [14]. This system is a library and set of API calls for building software dynamic translators on the x86. One such system built with DynamoRIO addresses code security. Unlike Strata, to the best of our knowledge, DynamoRIO was not designed with retargetability in mind. Another difference is that DynamoRIO is distributed as a set of binary libraries. The source code is available for Strata, making it possible to modify and experiment with the underlying infrastructure to implement new SDT systems.

To achieve high performance in a SDT system, it is important to reduce the overhead of the translation step. For a retargetable and flexible system like Strata, it can be all the more difficult to achieve good performance across a variety of architectures and operating systems. A number of SDT systems have tackled the overhead problem. For example, Shade [6] and the Embra [26] emulator use a technique called chaining to link together cache-resident code fragments to bypass translation lookups. This technique is similar to one of the overhead reduction techniques in Strata that links a series of fragments to avoid context switches. Other systems tackle the overhead of translation by doing the translation concurrently on a processor separate from the one running the application [24]. To address overhead issues, there are also efforts to develop hardware support for SDT [15]. One of the major sources of overhead in a system like Strata are indirect branches. Consequently both Dynamo [1] and Daisy [8] convert indirect branches to chains of conditional branches to improve program performance.

In Strata, the framework provides mechanisms for dealing with overhead in a retargetable fashion. These mechanisms are provided as target-independent services that consult machine-dependent routines. Strata tackles the overhead problem in two ways [16]. First, it chains code fragments together to minimize the number of builder re-enters. Second, Strata reduces the expense of indirect branches by caching previous translations of indirect branch target addresses. Strata's mechanisms for reducing overhead are target independent with support from target-dependent routines.

In addition to the three targets mentioned in this paper, Strata has also been ported to the SimpleScalar/PISA architecture in order to study the interactions between software dynamic translators and hardware.

Regretably, a complete discussion of this work is beyond the scope of this paper.

6. Summary

This paper has described the architecture of a software dynamic translation system, called Strata. To address the difficulty of initiating a new project in software dynamic translation, we designed Strata as a flexible and retargetable system. This paper described our experience retargeting Strata to the SPARC, MIPS, and x86 instruction set architectures. Each implementation was achieved with relatively modest effort, but each implementation had to deal with different target-machine idiosyncrasies. We expect that future retargets should take even less effort. The paper also described our experience with using Strata to develop two new SDT systems—one for safe execution of untrusted binaries and one for fast prototyping of architecture simulators. Other groups are using Strata to develop new applications of SDT technology, including software testing, architecture performance evaluation, and prototyping code optimizations. Reports from these groups indicate that Strata is meeting their needs and is a flexible and general system.

7. Acknowledgments

This work was supported in part by an Intel Graduate Fellowship, NSF grants EIA-0072043, ACI-0203956, and DARPA contract #. We would like to thank Kevin Skadron for many productive discussions during Strata's early development, and Kevin Hirst for porting Strata to the SimpleScalar PISA instruction set architecture. Finally, the first author wishes to thank the telematics research group at the University of Goettingen for providing the collegial environment in which work on this paper was finished.

8. References

- [1] BALA, V., DUESTERWALD, E., AND BANERJIA, S. Dynamo: A transparent dynamic optimization system. In *SIGPLAN '00 Conference on Programming Language Design and Implementation* (2000), pp. 1–12.
- [2] BRUENING, D., DUESTERWALD, E., AND AMARASINGHE, S. Design and implementation of a dynamic optimization framework for windows. In *Proceedings of the ACM Workshop on Feedback-Directed and Dynamic Optimization FDDO-4* (2001).
- [3] CHEN, W.-K., LERNER, S., CHAIKEN, R., AND GILLIES, D. Mojo: A dynamic optimization system. In *Proceedings of the ACM Workshop on Feedback-Directed and Dynamic Optimization FDDO-3* (2000).

- [4] CHERNOFF, A., HERDEG, M., HOOKWAY, R., REEVE, C., RUBIN, N., TYE, T., YADAVALLI, S. B., AND YATES, J. FX132: A profile-directed binary translator. *IEEE Micro* 18, 2 (Mar./Apr. 1998), 56–64. Presented at Hot Chips IX, Stanford University, Stanford, California, August 24–26, 1997.
- [5] CIFUENTES, C., LEWIS, B., AND UNG, D. Walkabout—a retargetable dynamic binary translation framework. In *Proceedings of the 2002 Workshop on Binary Translation* (2002).
- [6] CMELIK, B., AND KEPPEL, D. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems* (May 1994), pp. 128–137.
- [7] DITZEL, D. R. Transmeta’s Crusoe: Cool chips for mobile computing. In *Hot Chips 12: Stanford University, Stanford, California, August 13–15, 2000* (1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2000), IEEE, Ed., IEEE Computer Society Press.
- [8] EBCIOGLU, K., AND ALTMAN, E. DAISY: Dynamic compilation for 100% architectural compatibility. In *24th Annual International Symposium on Computer Architecture* (1997), pp. 26–37.
- [9] GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 1996 USENIX Security Symposium* (1996).
- [10] HENNESSY, J., AND PATTERSON, D. *Computer Architecture: A Quantitative Approach, 2nd ed.* Morgan Kaufmann Publishers Inc., Palo Alto, CA 94303, 1995.
- [11] INTEL. *Intel Architecture Software Developer’s Manual*, 1999. Instruction Set Reference.
- [12] JONES, M. B. Interposition agents: Transparently interposing user code at the system interface. In *Symposium on Operating Systems Principles* (1993), pp. 80–93.
- [13] KESSLER, P. B. Fast breakpoints. design and implementation. *ACM SIG-PLAN Notices* 25, 6 (June 1990), 78–84.
- [14] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. Secure execution via program shepherding. In *11th USENIX Security Symposium* (2002).
- [15] MERTEN, M. C., TRICK, A. R., BARNES, R. D., NYSTROM, E. M., GEORGE, C. N., GYLLENHAAL, J. C., AND MEI W. HWU, W. An architectural framework for runtime optimization. *IEEE Transactions on Computers* 50, 6 (2001), 567–589.
- [16] SCOTT, K., AND DAVIDSON, J. Low-overhead software dynamic translation. Tech. Rep. CS-2001-18, 2001.
- [17] SCOTT, K., AND DAVIDSON, J. Strata: A software dynamic translation infrastructure. In *IEEE Workshop on Binary Translation* (2001).
- [18] SCOTT, K., AND DAVIDSON, J. Safe virtual execution using software dynamic translation. In *Annual Computer Security Applications Conference* (2002).
- [19] SRIVASTAVA, A., EDWARDS, A., AND VO, H. Vulcan: Binary translation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, Apr. 2001.
- [20] TAMCHES, A., AND MILLER, B. P. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI-99)* (Berkeley, CA, Feb. 22–25 1999), Usenix Association, pp. 117–130.
- [21] TAMCHES, A., AND MILLER, B. P. Using dynamic kernel instrumentation for kernel and application tuning. *The International Journal of High Performance Computing Applications* 13, 3 (Fall 1999), 263–276.
- [22] TRÖGER, J., AND GOUGH, J. Fast dynamic binary translation—the yirt-ma framework. In *Proceedings of the 2002 Workshop on Binary Translation* (2002).
- [23] UNG, D., AND CIFUENTES, C. Machine-adaptable dynamic binary translation. In *Proceedings of the ACM Workshop on Dynamic Optimization Dynamo ’00* (2000).
- [24] VOSS, M., AND EIGENMANN, R. A framework for remote dynamic program optimization. In *Proceedings of the ACM Workshop on Dynamic Optimization Dynamo ’00* (2000).
- [25] WEAVER, D. L., AND GERMOND, T. *The SPARC Architecture Manual Version 9*. Prentice-Hall PTR, Upper Saddle River, NJ 07458, USA, 1994.
- [26] WITCHEL, E., AND ROSENBLUM, M. Embra: Fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (May 1996), pp. 68–79.