Strata: A Software Dynamic Translation Infrastructure

Kevin Scott
Department of Computer Science
University of Virginia
kscott@cs.virginia.edu

Jack Davidson
Microsoft Research
Microsoft Corporation
jwd@microsoft.com

Abstract

Software dynamic translation is the alteration of a running program to achieve a specific objective. For example, a dynamic optimizer uses software dynamic translation to modify a running program with the objective of making the program run faster. In addition to its demonstrated utility in dynamic optimizers, software dynamic translation also shows promise for producing applications that are adaptable, secure, and robust. In this paper, we describe the design and implementation of an extensible, retargetable infrastructure to facilitate research in applications of software dynamic translation technology. The infrastructure, called Strata, provides the software dynamic translator implementor with a virtual machine model that can be extended to implement specific software dynamic translation functionality. To illustrate the use of Strata to build client applications, the paper describes the Strata implementation of a two dynamic safety checkers and a dynamic instruction scheduler.

1. Introduction

Software dynamic translation is the alteration of a running program to achieve a specific objective. Despite apparent differences, software dynamic translation systems are fundamentally similar. Software dynamic translation systems 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 a program's instructions before they are run on the host CPU.

Software dynamic translation gives system designers unprecedented flexibility to control and modify a program's execution. This flexibility allows software dynamic translation to be used to achieve a variety of objectives not easily obtainable 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 [Ditzel 2000]. Similarly, the UQDBT system dynamically translates Intel IA-32 binaries to run on SPARC-based processors [Ung and Cifuentes 2000].

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 [Cmelik and Keppel 1994] uses software dynamic translation to implement high-performance instruction set simulators. Embra [Witchel and Rosenblum 1996] uses software dynamic translation to implement a high-performance operating system emulator. Dynamo [Bala et al. 2000] uses software dynamic translation to improve the performance of native binaries, and Daisy [Ebcioglu and Altman 1997] uses software dynamic translation to evaluate the performance of novel VLIW architectures and accompanying optimization techniques. The ParaDyn [Miller et al. 1995] and Vulcan [Srivastava et al. 2001] systems use software dynamic translation to insert performance monitoring instrumentation into running programs. There are also many other commercial and research systems that use software dynamic translation to meet their requirements.

Unfortunately, many existing software dynamic translation systems were designed to implement a specific type of software dynamic translator for specific architectures. Flexibility and retargetability were typically not design goals of these systems. For example, the Daisy system was designed to translate code for a VLIW to the PowerPC [Ebcioglu and Altman 1997]. It is unclear what effort would be required to retarget Daisy to emit code for another processor. Similarly, the DynInst API [Buck and Hollingsworth 2000] and Vulcan [Srivastava et al. 2001] can both be used to insert code into a running program. The code inserted by these tools typically serves to collect information about the performance of the running program. It is unclear if either tool would be suitable for building software

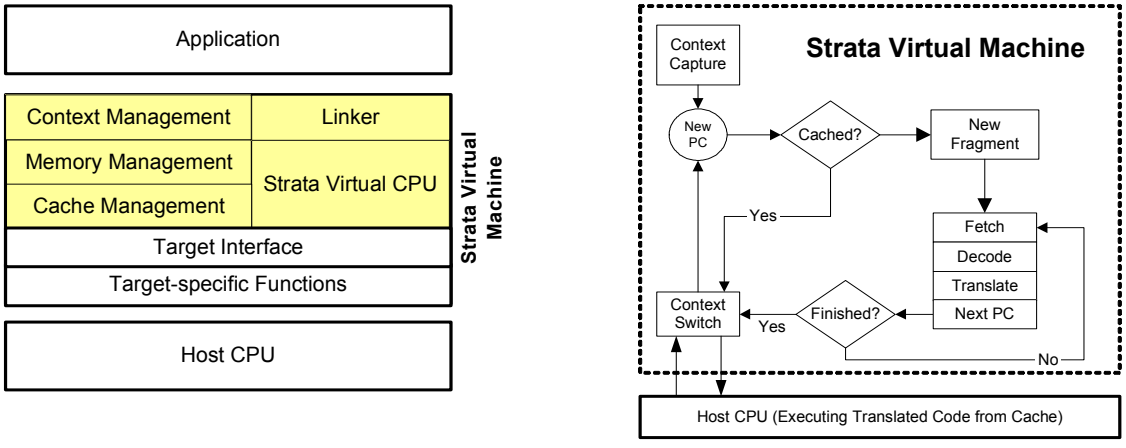
dynamic translators where profiling is not the primary objective.

In this paper, we describe the architecture of Strata, a software dynamic translation infrastructure designed to address the concerns of flexibility and retargetability. Our goal was an infrastructure that provides extensible facilities to enable rapid experimentation with software dynamic translation for a variety of purposes across a range of architectures and operating systems. The paper describes Strata's overall architecture with a focus on the extensibility features that allow users to tailor Strata to meet their objectives. We also describe how Strata is retargeted to another architecture. Currently, Strata runs on SPARC and MIPS-based platforms (running Solaris and IRIX operating systems, respectively). To illustrate Strata's use, we briefly describe the construction of three Strata-based applications. Two applications perform dynamic safety checks—one prevents stack-smashing attacks, the other one prevents execution of unauthorized system calls. The third application is a dynamic instruction scheduler.

2. DESIGN of STRATA

Strata is an infrastructure for building software dynamic translators. Strata is organized as a virtual machine that mediates execution of an application's instructions. Specific software dynamic translators are implemented in Strata by extending this virtual machine to perform new functions. Making virtual machine extensions easy to write is Strata's primary design goal. To achieve this goal 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 (see Figure 1a.) 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.

Later in this section we will describe the Strata common services and the target interface in greater detail. For now we focus on the basic operation of the Strata virtual machine. The Strata VM mediates application execution by examining and translating instructions before they execute on the host CPU. Figure 1b shows the architecture of the Strata VM. The Strata VM is first entered by capturing and saving the application context (e.g., PC, condition codes, registers.) The Strata VM begins processing 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.



(a) (b)

Figure 1. Strata architecture. (a) The Strata VM sits between the application and the host CPU, and is comprised of a set of target-independent *common services* (shaded boxes), a set of *target-specific functions*, and a *target-interface* through which the two communicate. (b) Strata operates by fetching, decoding, and translating an application's instructions into a cache. The VM components depicted as rectangles are target-specific functions invoked through the target interface.

If there is no cached translation for the next application instruction, the Strata VM allocates storage in the cache 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 case, when the end-of-fragment condition is met, a *context switch* restores the application context and the newly translated fragment is executed.

Figure 1a shows the components of the Strata virtual machine. Strata consists of 5000 lines of C code, roughly half of which is target-specific. Shaded boxes show the Strata common services which comprise the remaining half of the Strata source. The Strata common services are target-independent and implement functions that may be useful in a variety of Strata-based dynamic translators. Features such as context management, memory management, and the Strata virtual CPU will most likely be required by any Strata-based dynamic translator. The cache manager and the linker can be used to improve the performance of Strata-based dynamic translators, and are detailed in other work [Scott and Davidson 2001].

2.1. Target Interface

Target-specific functions and target-independent common services communicate with one another through the target interface. The target interface allows the Strata VM to be easily reconfigured to include new software dynamic translation functionality, and makes retargeting to a new platform a straightforward process. When adding new functionality to Strata for an existing target, users will *override* one or more functions in the target interface with their own functions that perform the desired function. When retargeting Strata to a new platform, all of the functions in the target interface must be implemented[1].

Table 1 lists the 19 target-specific functions required by the target interface along with a brief description of each. Less than half of the functions in the target interface are used to read and translate application instructions. The remaining functions implement context management (`capture`, `exec`), i-cache management (`flush`), low-level memory management (`alloc`, `alloc_exec`), fragment cache customization (`begin_fragment`, `end_fragment`, `emit`), low-level linking (`patch`), and initialization customization (`init`).

To illustrate how the target interface is used to implement a new software dynamic translator, we consider the task of preventing stack-smashing attacks. Stack-smashing attacks take advantage of unsafe buffer manipulation functions (e.g., `strcpy` from the C standard library) to copy, and subsequently execute, malicious code from the application stack. The malicious code is executed with the privileges of the user running the program, and in many cases can lead to serious security compromises on affected systems [Larochelle and Evans 2001, Cowan et al. 1998].

A simple way to prevent stack-smashing attacks is to make the application stack non-executable. For a number of reasons which we will not detail here, this simple mechanism is rarely used in production systems.[2] Fortunately, it is an almost trivial matter to implement a software dynamic translator in Strata that prevents execution of stack code. The first step is to override the fetch function in the default SPARC target interface with a custom fetch. The following code accomplishes this.

```
TI = SPARC_target_interface;
TI.fetch = my_fetch;
```

The custom fetch shown in Table 2 checks the PC against the stack boundaries and terminates program execution if the instruction being fetched is on the stack. Otherwise, the default SPARC fetch function is called.

Another example of a Strata-based software dynamic translator is a simple fragment instruction scheduler. Such an instruction scheduler could be used to improve application performance by minimizing visible delays associated with long latency operations. Dynamic instruction scheduling allows code originally scheduled for one microarchitecture to be rescheduled to improve performance on another. Furthermore, performing instruction scheduling dynamically on fragments may have advantages over static approaches

1. In some cases this might not be entirely true. The retargeter may be able to copy the target-specific functions from another platform and modify them for the new target. For architectures that have many similarities, as is the case with many commercial RISC platforms, the modifications required can be quite straightforward.

2. On many platforms the operating system maps the stack onto an executable segment and does not allow the application programmer to choose any other policy.

Table 1. Target interface functions.

Function	Description
fetch	Fetches an application instruction from the current PC.
nextPC	Computes the PC of the next instruction to process.
decode	Decode enough of the instruction to dispatch to the right translator.
translate_PCrel_branch	Translate a PC relative branch.
translate_indir_branch	Translate a register-indirect (computed) branch.
translate_call	Translate a call.
translate_return	Translate a return.
translate_special	Translate a special instruction (e.g., traps, i-cache flushes, etc.)
translate_normal	Translate any instruction not handled by another translator.
begin_fragment	Called before the first instruction is emitted into a newly allocated fragment.
end_fragment	Called after the last instruction is emitted into a fragment.
emit	Emit an instruction into a fragment.
flush	Synchronize the host CPU's i-cache and memory.
exec	Restore application context & execute the specified fragment.
alloc_exec	Allocate a chunk of contiguous, executable memory.
alloc	Allocate a chunk of contiguous memory.
patch	Patch an address embedded in fragment cache instructions.
capture	Capture and save the application context.
init	Target-specific initialization.

Table 2. Custom fetch that prevents execution of code on the stack.

```
insn_t my_fetch (iaddr_t PC) {
    if (PC >= stack_low && PC <= stack_high) {
        strata_fatal("Attempted to execute code on the stack");
    } else {
        return sparc_fetch(PC);
    }
}
```

since fragments can span function calls (in fact any PC-relative, unconditional control transfer) thus exposing more instructions to the scheduler.

To implement a dynamic instruction scheduler in Strata, we first override the `end_fragment` function in the default SPARC target interface with a custom `end_fragment` function. The following code accomplishes this.

```
TI = SPARC_target_interface;
TI.end_fragment = my_end_fragment;
```

The custom `end_fragment` function in Table 3 completes the implementation of the dynamic fragment instruction scheduler. This function calls a list scheduler (`sparc_list_schedule`) on the newly formed fragment before the fragment is first executed. After the fragment's instructions have been scheduled, the default SPARC `end_fragment` function is called.

These two examples illustrate the power of Strata's organization. The target interface allows useful new dynamic translators to be implemented without requiring major changes to Strata, or requiring the programmer to understand the inner workings of Strata.

Table 3. Custom `end_fragment` that schedules instructions in a fragment.

```
void my_end_fragment (strata_fragment *frag) {
    sparc_list_schedule(frag);
    sparc_end_fragment(frag);
}
```

2.2. Common Services

The Strata common services are target-independent routines which are useful, and sometimes essential, to a variety of dynamic translators. The Strata common services consist of 2500 lines of C code in version 1.0 of Strata. Common services include memory management, code cache management, application context management, a dynamic linker, and a virtual CPU that mimics standard hardware fetch/decode/execute engines.

As many programs do, Strata operates on a non-trivial number of dynamically allocated data structures. Since Strata typically runs in the same address space as the application it is executing, it is beneficial for Strata to manage its own heap. Strata has special dynamic allocation and deallocation patterns that we can take advantage of to improve the performance of heap memory management. The Strata memory manager uses the `alloc` function supplied by the target interface to allocate memory in large chunks. Strata organizes these bulk-allocated memory chunks into arenas [Hanson 1997]. An arena is a memory management structure that permits many dynamically allocated objects to be discarded at once, thus freeing a program from the error-prone and overhead-laden task of releasing allocations individually. Each arena in Strata contains dynamically allocated objects with similar lifetimes. Dynamically allocated objects tend to be live through an entire program execution, through the lifetime of the code cache, or through a single context switch-in/out of the Strata virtual machine. As such, the Strata memory manager organizes the heap into three arenas.

Code cache management provided as a Strata common service is very simple. Fragments are allocated sequentially within a contiguous region of memory. The contiguous region of memory must be executable and is allocated by the `alloc_exec` function in the target interface. When no more space is available in the code cache for new fragments, the entire contents of the cache are discarded. This is the cache management policy used by the Dynamo dynamic optimizer [Bala et al. 2000] and the Embra OS emulator [Witchel and Rosenblum 1996]. The advantages of such a policy are simplicity of implementation and fast fragment allocation and deallocation. The disadvantage is that when the executed portions of the application binary do not entirely fit within the code cache, the simple eviction policy results in potentially useful fragments being discarded. In practice, we have observed that sizing the fragment cache large enough to hold all executed portions of an application results only in a negligible increase in the application's maximum resident set size [Scott and Davidson 2001]. Further, both Dynamo and Embra achieve reasonable performance even when there are many potentially wasteful evictions. A detailed evaluation of software dynamic translation cache management policies is beyond the scope of this paper.

The software dynamic translator and the application being dynamically translated may be viewed as two coroutines. Both execute on the native CPU and most likely use some of the same CPU resources. The CPU resources that may be used by both the dynamic translator and the application are known as the application context. It is the job of the Strata application context manager to save and restore this context when the application control switches to and from Strata.

In Strata's basic mode of operation, control is transferred to the fragment builder after each application branch executes. A large portion of these context switches can be eliminated by linking fragments together as they materialize into the code cache. For instance, when one or both of the destinations of a PC-relative conditional branch materialize in the code cache, the conditional branch trampoline can be rewritten to transfer control directly to the appropriate fragment cache locations rather than performing a context switch and control transfer to the fragment builder. Linking together cached code translations is a technique employed by many software dynamic translators. The Shade simulator, Embra emulator, and Dynamo dynamic optimizer all employ this technique. The Strata dynamic linker is the service which performs linking of translated fragments.

The Strata virtual CPU performs actual fragment formation. The virtual CPU mimics the organization of a real CPU. Instructions are fetched from the application, partially decoded, and then translated. These functions are supplied by the target interface, but their sequencing is controlled by the virtual CPU.

3. Strata At Work: A System Call Monitor

In Section 2 we sketched two trivial examples in order to demonstrate the process one uses to write a

Strata-based software dynamic translator. In this section we present a complete Strata-based software dynamic translator that performs a useful function. We also present an experimental evaluation of this dynamic translator's performance.

In the era of ubiquitous networking managing untrusted binaries has become an important security concern. Untrusted binaries are programs which a user might download over the network onto their local machine. Users can not be sure that such a binary does not misappropriate resources on their local machine either deliberately for mischief's sake, or accidentally through a bug in the program[1]. A program may waste CPU time by doing useless operations, but frequently, the misappropriations of resources that cause harm occur through execution of system calls. In fact programs that are susceptible to stack-smashing fall into this category: the malicious code executed from the stack almost always uses system calls to gain unauthorized access to system resources.

Strata can be used to implement a dynamic translator that enforces a predefined system call utilization policy. For example, the policy may specify that an untrusted program may not open network sockets, or may only write to particular file descriptors, or may not perform an `exec()`. We can override the `translate_special` function in the target interface to recognize system calls and insert code before each that checks system call parameters for adherence to the policy. If the policy is violated, the offending system call is not executed, and the program may be terminated or other appropriate action taken.

If the overhead of the system call policy safety checker is excessively high, then the applicability of this technique in real systems may be limited. In Figure 2, the column labeled "Baseline" shows the slowdown due to executing each program under Strata. The column labeled "System Call" shows the slowdown due to executing each program under a Strata that has been extended to prohibit `exec()` system calls from executing. This policy is sufficiently powerful to prevent two known stack-smashing attacks on privileged Solaris programs [Schipor 1996, McGann 1998]. The slowdowns associated with this safety checker average 1.32x which is not noticeably different from the baseline Strata.

As a point of reference, the Janus project proposed a sand-boxing technique that enforces a predefined system call utilization policy [Goldberg et al. 1996]. In Janus system calls are dynamically intercepted using an OS system call trace facility. Their system is transparent (as is ours) and performs sand-boxing at very low overhead. They report lower overheads than our Strata-based system call monitor. To achieve low overhead, however, their system refrains from monitoring frequently executed system calls (e.g., `write()`). Furthermore, they rely on a nonstandard, low overhead system call tracing facility. In contrast, our Strata-based system call policy safety checker does not rely on special OS system call tracing facilities and it incurs no additional performance penalty when monitoring frequently executed system calls.

1. It is not obvious that the problem diminishes even when users have access to source code and can compile programs themselves.

4. Summary

This paper has provided an overview of the architecture of Strata, an infrastructure for building software dynamic translators. Strata's design was driven by our desire to build an infrastructure that was extensible and retargetable. Extensibility yields an infrastructure that supports building a variety of client applications that use software dynamic translation to achieve their objectives. Retargetability yields an infrastructure that can be used across a range of architectures and operating systems.

While our work with Strata is less than a year old, our initial experience using Strata to build several novel client applications has been positive. We are confident that Strata's basic design provides the necessary basic services as well as the ability for a user to extend these services to build SDT-based applications with modest effort. The paper described the construction of two very different types of Strata client applications: two safety checkers that prevent buffer-overrun exploits and execution of unauthorized system calls, and a dynamic instruction scheduler. As we continue to build other software dynamic translator-based applications and gain more experience, we expect to refine our design and implementation.

We have less confidence regarding Strata's retargetability. To date Strata has been targeted to only two architectures—SPARC and MIPS (the MIPS target was done by another group). While the preliminary results have been good, our extensive experience building retargetable software is that one needs to handle at least three to four machines (including the IA-32!) and use several different compilers and operating systems (including Windows!) to discover all the machine and operating system dependencies. Consequently, we have no doubt that Strata's implementation will continue to evolve as we accommodate additional architectures and operating systems.

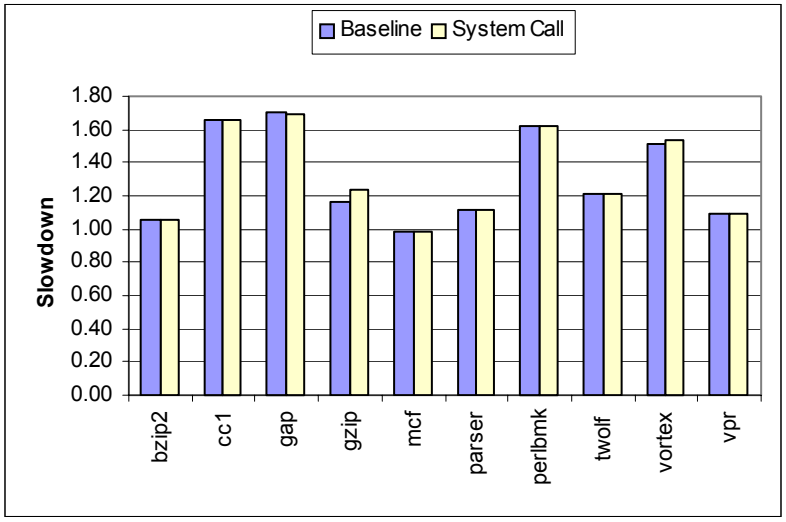


Figure 2. Performance of Strata-based system call monitor.

5. References

BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: A transparent dynamic optimization system. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*. 1–12.

BUCK, B. AND HOLLINGSWORTH, J. K. 2000. An API for runtime code patching. *The International Journal of High Performance Computing Applications 14,* 4 (Winter), 317–329.

CMELIK, R. F. AND KEPPEL, D. 1994. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. ACM Press, New York, NY, USA, 128–137.

COWAN, C., PU, C., MAIER, D., HINTON, H., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. 1998. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 1998 USENIX Security Symposium*.

DITZEL, D. R. 2000. Transmeta's Crusoe: Cool chips for mobile computing. In *Hot Chips 12: Stanford University, Stanford, California, August 13–15, 2000*, IEEE, Ed. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA.

EBCIOGLU, K. AND ALTMAN, E. 1997. DAISY: Dynamic compilation for 100% architectural compatibility. In *24th Annual International Symposium on Computer Architecture*. 26–37.

GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. 1996. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 1996 USENIX Security Symposium*.

HANSON, D. R. 1997. *C Interfaces and Implementations: Techniques for Creating Reusable Software*. Addison-Wesley, Reading, MA, USA.

LAROCHELLE, D. AND EVANS, D. 2001. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*.

MCGANN, S. 1998. The solaris ufsdump buffer-overrun exploit. `http://www.insecure.org/sploits/Solaris.ufsdump.ufsrestore.html`.

MILLER, B. P., CALLAGHAN, M. D., CARGILLE, J. M., HOLLINGSWORTH, J. K., IRVIN, R. B., KARAVANIC, K. L., KUNCHITHAPADAM, K., AND NEWHALL, T. 1995. The Paradyn parallel performance measurement tool. *Computer 28,* 11 (Nov.), 37–46.

SCHIPOR, C. 1996. The solaris eject buffer-overrun exploit. `http://www.insecure.org/sploits/solaris.eject.html`.

SCOTT, K. AND DAVIDSON, J. W. 2001. Low-overhead software dynamic translation. In submission.

SRIVASTAVA, A., EDWARDS, A., AND VO, H. 2001. Vulcan: Binary translation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research. Apr. 2001.

UNG, D. AND CIFUENTES, C. 2000. Machine-adaptable dynamic binary translation. In *Proceedings of the ACM Workshop on Dynamic Optimization Dynamo '00*. 41–51.

WITCHEL, E. AND ROSENBLUM, M. 1996. Embra: Fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 68–79.