Low-Overhead Software Dynamic Translation

Technical Report CS-2001-18

Kevin Scott, Jack W. Davidson, and Kevin Skadron

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22904

{kscott,jwd,skadron}@cs.virginia.edu

Abstract

Software dynamic translation (SDT) is a technology that allows programs to be modified as they are running. The overhead of monitoring and modifying a running program's instructions is often substantial in SDT. As a result SDT can be impractically slow, especially in SDT systems that do not or can not employ dynamic optimization to offset overhead. This is unfortunate since SDT has obvious advantages in modern computing environments and interesting applications of SDT continue to emerge. In this paper we introduce two novel overhead reduction techniques that can improve SDT performance by a factor of three even when no dynamic optimization is performed. To demonstrate the effectiveness of our overhead reduction techniques, and to show the type of useful tasks to which low-overhead, non-optimizing SDT systems might be put, we implemented two dynamic safety checkers with SDT. These dynamic safety checkers perform useful tasks–preventing buffer-overrun exploits and restricting system call usage in untrusted binaries. Further their performance is similar to, and in some cases better than, state-of-the-art tools that perform the same functions without SDT.

1 INTRODUCTION

Software dynamic translation (SDT) is a technology that allows programs to be modified as they are running. SDT systems virtualize aspects of the host execution environment by interposing a layer of software between program and CPU. This software layer mediates program execution by dynamically examining and translating a program's instructions before they are run on the host CPU. Recent trends in research and commercial product deployment strongly indicate that SDT is a viable technique for delivering adaptable, high-performance software into today's rapidly changing, heterogeneous, networked computing environment.

SDT is used to achieve distinct goals in a variety of research and commercial systems. One of these goals is binary translation. Cross-platform SDT allows binaries to execute on non-native platforms. This allows existing applications to run on different hardware than originally intended. Binary translation makes introduction of new architectures practical and economically viable. Some popular SDT systems that fall into this category are FX!32 (which translates IA-32 to Alpha) [7], DAISY (which translates VLIW to PowerPC) [13], UQDBT (which translates IA-32 to SPARC) [19], and Transmeta's Code Morphing technology (which translates IA-32 to VLIW) [10].

Another goal of certain SDT systems is improved performance. Dynamic optimization of a running program offers several advantages over compile-time optimization. Dynamic optimizers use light-weight execution profile feedback to optimize frequent executed (hot) paths in the running program. Because they collect profile information while the program is running, dynamic optimizers avoid training-effect problems suffered by static optimizers that use profiles collected by (potentially non-representative) training runs. Furthermore, dynamic optimizers can continually monitor execution and reoptimize if the program makes a phase transition that creates new hot paths. Finally, dynamic optimizers can perform profitable optimizations such as partial inlining of functions and conditional branch elimination that would be too expensive to perform statically. Some SDT systems that perform dynamic optimization are Dynamo (which optimizes PA-RISC binaries) [4, 11], Vulcan (which optimizes IA-32 binaries) [17], Mojo (which optimizes IA-32 binaries) [6], DBT (which optimizes PA-RISC binaries) [12], and Voss and Eigenmann's remote dynamic program optimization system (which optimizes SPARC binaries using a separate thread for the optimizer) [20]. Some of the binary translators previously described also perform some dynamic optimization (e.g., DAISY, FX!32, and Transmeta's Code Morphing technology).

SDT is also a useful technique for providing virtualized execution environments. Such environments provide a framework for architecture and operating systems experimentation as well as migration of applications to different operating environments. The advantage of using SDT in this application area is that the simulation of the virtual machine is fast—sequences of virtual machine instructions are dynamically translated to sequences of host machine instructions. Examples of this application of SDT are Embra (which virtualizes the MIPS instruction set running on IRIX) [22], Shade (which runs on the SPARC and virtualizes both the SPARC and MIPS instruction sets) [8], VMware (which virtualizes either Windows or Linux) [16], and Plex86 (which virtualizes Windows for execution under Linux) [1].

Most of the preceding applications of SDT can benefit from reductions of dynamic translation overhead. Reducing overhead improves overall application performance, allows SDT systems to implement additional functionality (e.g., additional optimizations, more detailed profiling, etc.), and enables uses of SDT in new application areas. In this paper, we describe two novel techniques for reducing the overhead of SDT. Using Strata, a framework we designed for building SDT systems, we performed experiments to identify and measure sources of SDT overhead. We observed that SDT overhead stems from just a few sources, specifically the handling of indirect control transfers. Using our measurements as a guide, we implemented two novel techniques for reducing SDT overhead associated with indirect control transfers. The resulting improvement in overhead for non-optimizing SDT averages a factor of three across a broad-range of benchmark programs, and in some cases completely eliminates the overhead of non-optimizing SDT.

To demonstrate that low-overhead SDT can be applied to new and interesting application areas, we implemented two dynamic safety-checking applications. One safety checker prevents buffer-overflow attacks. Another safety checker prevents unauthorized system calls. Both safety checkers were implemented using Strata, a framework that we have designed for building efficient SDT applications. Using our overhead reduction techniques, the performance of the safety checkers is similar to, and in some cases better than, state-of-the-art tools that perform the same functions without SDT.

This paper makes two important contributions. First we show that SDT overhead can be substantially reduced by careful handling of indirect control transfers. These overhead reduction techniques are applicable to, and should improve the performance of, a wide variety of SDT systems, including dynamic optimizers. Second we show that useful and efficient applications can be built with non-optimizing SDT when our overhead reduction techniques are used.

The remainder of the paper is organized as follows. Section 2 briefly describes the Strata framework. Section 3 discusses SDT overhead. Section 4 discusses the SDT safety checkers that we implemented. Section 5 discusses related work and Section 6 presents our summary and conclusions.

2 STRATA

Strata is an infrastructure for building software dynamic translators. To realize a specific dynamic translator Strata basic services are extended to provide the desired functionality. The Strata basic services implement a very simple dynamic translator that mediates execution of native application binaries with no visible changes to application semantics, and no aggressive attempts to optimize application performance.

Figure 1 shows the high-level architecture of Strata. Strata provides the functionality in the prior description through a set of retargetable, extensible, SDT services. These services include memory management, fragment cache management, application context management, a dynamic linker, and a fetch/decode/translate engine.

Strata has two mechanisms for gaining control of an application. The application binary can be rewritten to replace the call to `main()` with a call to a Strata entry point. Alternately, the programmer can manually initiate Strata mediation by placing a call to `strata_start()` in their application. In either case, entry to Strata saves the application state, and invokes the Strata component known as the fragment builder. The fragment builder takes the PC of the next instruction that the program needs to execute, and if the instruction at that PC is not cached, the fragment builder begins to form a sequence of code called a fragment. Strata attempts to make these fragments as long as possible. To this end, Strata inlines unconditional PC-relative control transfers[1] into the fragment being constructed. In this mode of operation, each fragment is terminated by a conditional or indirect control transfer instruction[2]. However, since Strata needs to maintain control of program execution, the control transfer instruction is replaced with a *trampoline* that arranges to return control to the Strata fragment builder. Once a fragment is fully formed, it is placed in the fragment cache.

The transfers of control from Strata to the application and from the application back to Strata are called *context switches* (see Figure 1). On context switch into Strata via a trampoline, the current PC is looked up in a hash table to determine if there is a cached fragment corresponding to the PC. If a cached fragment is found, a context switch back to the application occurs. As will be discussed in Section 3, these context switches are a large component of SDT overhead.

1. On many architectures, including the SPARC, this includes unconditional branches and direct procedure calls.
2. The dynamic translator implementor may choose to override this default behavior and terminate fragments with instructions other than conditional or indirect control transfers.

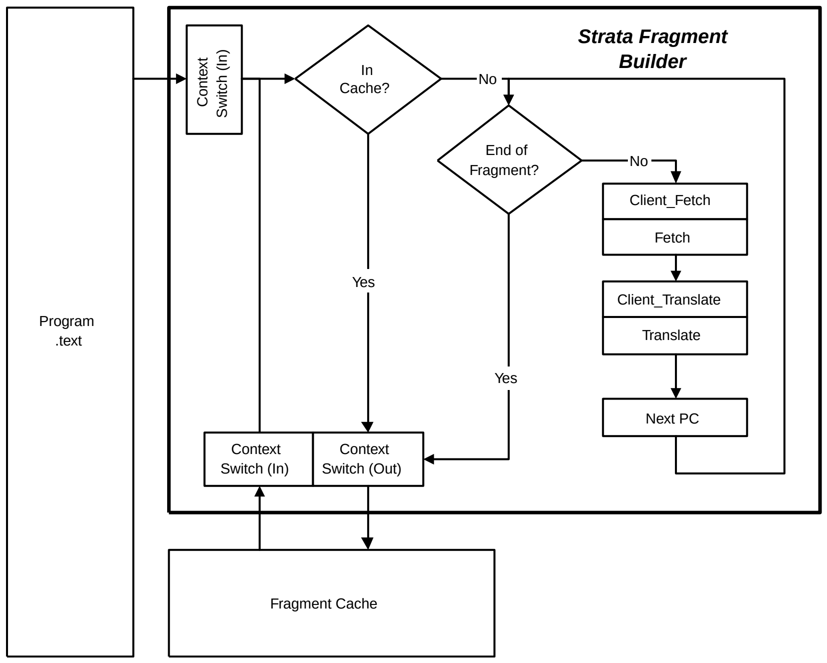

Figure 1. Architecture of Strata.

3 SOFTWARE DYNAMIC TRANSLATION OVERHEAD

Overhead in SDT systems can degrade overall system performance substantially. This is particularly true of dynamic translators which do not perform code optimizations to offset dynamic translation overhead. Overhead in software dynamic translators can come from time spent executing instructions not in the original program, from time lost due to the dynamic translator undoing static optimizations, or from time spent mediating program execution. For cross-platform dynamic translators, overhead may arise from inefficient translations between source and target instruction sequences. This is a source of overhead that we will not be concerned with in this paper since we are focusing on native SDT systems where the source and target architectures are the same.

To characterize overhead in such an SDT, we conducted a series of experiments to measure where our SDT systems spend their time. Our experiments were conducted on an unloaded SUN

400MHz UltraSPARC-II with 1GB of main memory. The basic Strata dynamic translator described in Section 2 was used for all experiments. This basic translator does no optimization. All experiments were performed using a 4MB fragment cache which is sufficiently large to hold all executed fragments for each of the benchmarks. Benchmark programs from SPECint2K[1] were compiled with Sun's C compiler version 5.0 with aggressive optimizations (-xO4) enabled. The resulting binaries were executed under the control of a Strata dynamic translator. We used the SPECint2K training inputs for all our measurement runs. [18]

In Strata's basic mode of operation, a context switch occurs after each fragment executes. A large portion of these context switches can be eliminated by linking fragments together as they materialize into the fragment cache. For instance, when one or both of the destinations of a PC-relative conditional branch materialize in the fragment 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.

Figure 2 shows the slowdown of our benchmark programs when executed under Strata with and without fragment linking. Slowdowns are relative to the time to execute the application directly on the host CPU. Without fragment linking, we observed very large slowdowns—an average of 22.9x across all benchmarks. With fragment linking, the majority of context switches due to executed conditional branches are eliminated. The resulting slowdowns are much lower, but still impractically high—an average of 4.1x across all benchmarks,.

1. The benchmarks eon and crafty were not used in our experiments. We chose to eliminate these two programs since eon is a C++ application and crafty requires 64-bit C longs, neither of which were supported by the compiler and optimization settings used for the rest of the benchmarks.

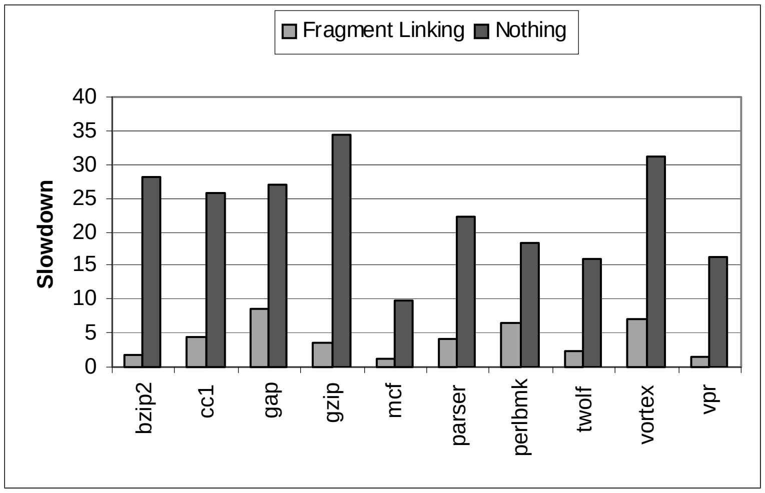

Figure 2. Slowdown with simple overhead reduction.

The majority of the remaining overhead is due to the presence of indirect control transfer instructions. Because the target of an indirect control transfer is only known when the branch executes, Strata cannot link fragments ending in indirect control transfers to their targets. As a consequence each fragment ending in an indirect control transfer must save the application context and call the fragment builder with the computed branch-target address. The likelihood is very high that the requested branch target is already in the fragment cache, so the builder can immediately restore the application context and begin executing the target fragment. The time between reaching the end of the indirect control transfer and beginning execution at the branch target averages about 250 cycles on the SPARC platform that we used for our experiments. This is a large penalty to pay in programs which execute large numbers of indirect control transfer instructions.

On the SPARC, indirect control transfers fall into two categories—function call returns and other indirect branches. Figure 3 shows the number of context switches to Strata due to either returns or other indirect branches. It is clear from this figure that the mix of indirect control transfers is highly application dependent. In the benchmarks gzip, parser, vpr, and bzip2, almost all indirect control transfers executed are returns with a few non-return indirect branches. In contrast cc1, perlbmk, and gap execute a sizeable fraction of indirect control transfers that are not returns. These applications contain many C *switch* statements that the Sun C compiler implements using indirect branches through jump tables. In the remaining applications, mcf, vortex and twolf, most control transfers are returns and a very small portion are indirect branches.

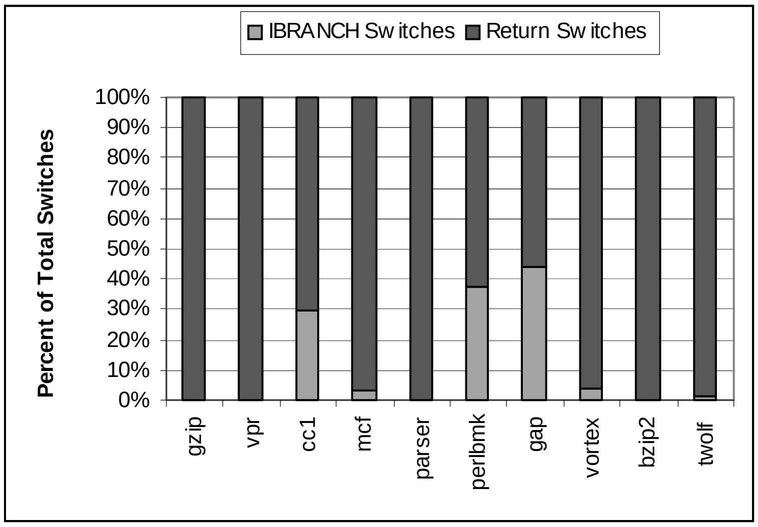

Figure 3. Causes of context switching using fragment linking, and no other overhead reduction.

To improve Strata overhead beyond the gains achieved from fragment linking we must either find a way to reduce the latency of individual context switches to Strata, or we must reduce the overall number of switches due to indirect control transfers. The code which manages a context switch is highly-tuned, hand-written assembler language. It is very unlikely that we can significantly reduce execution time of this code below the current 250 cycles. However, we have developed two novel and highly effective techniques for reducing the number of context switches forced by indirect control transfers.

3.1 Indirect Branch Translation Caching

The first technique that we propose for reducing the number of context switches due to indirect control transfer is the indirect branch translation cache (IBTC). An IBTC is a small, direct-mapped cache that maps branch-target addresses to their fragment cache locations. We can choose to associate an IBTC with every indirect control transfer instruction or just with non-return control transfer instructions. An IBTC in many respects is like the larger lookup table that the fragment builder uses to locate fragments in the fragment cache. However, an IBTC is a much simpler structure, and consequently much faster to consult. An IBTC lookup requires a few instructions which can be inserted directly into the fragment, thereby avoiding a full context switch.

The inserted code saves a portion of the application context and then looks up the computed indirect branch target in the appropriate IBTC. If the branch target matches the tag in the IBTC (i.e., a IBTC hit), then the IBTC entry contains the fragment cache address to which the branch target has been mapped. The partial application context is restored, and control is transferred to the branch target in the fragment cache. An IBTC hit requires about 15 cycles to execute, an order of magnitude faster than a full context switch. On an IBTC miss, a full context switch is performed and the Strata fragment builder is invoked. In addition to the normal action taken on a context switch, the address that produced the miss replaces the old IBTC entry. Subsequent branches to this location should hit in the IBTC.

Figure 4 shows the miss rates for various IBTC sizes. The left chart shows miss rates when only non-return indirect control transfers are handled with IBTCs. The right chart shows miss rates when all indirect control transfers are handled with IBTCs. When returns are included, the higher volume of indirect control transfers result in capacity and conflict misses that push the overall IBTC miss rate higher. Not surprisingly, miss rates are also higher when using smaller IBTC sizes. Generally once IBTC size exceeds 256 entries improvements in miss rate begin to level off for most programs.

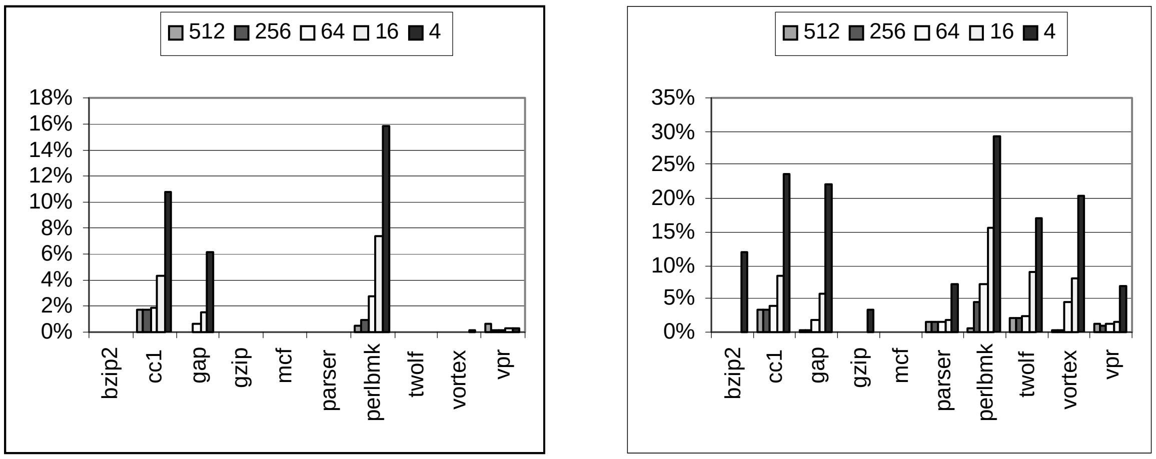

Figure 4. IBTC miss rate per benchmark. The chart on the left shows IBTC miss rate when the IBTC handles non-return indirect branches. The chart on the right shows IBTC miss rate when it must handle all indirect branches, including returns.

The performance benefits from using an IBTC are substantial. In Figure 5, the white bar shows application slowdowns when using fragment linking and 512-entry IBTCs to handle all indirect control transfers, including returns (the other results contained in Figure 5 will be discussed in Section 3.2). The average slowdown across all benchmarks is 1.7x which is significantly better than the average 4.1x slowdown observed with fragment linking alone. As we would expect, the largest slowdowns are observed in programs with large numbers of frequently executed switches such as perlbmk, cc1, and gap.

3.2 Fast Returns

Even though the IBTC mechanism yields low miss rates, due to the large percentage of executed returns (see Figure 3) and the overhead of the inserted instructions to do the IBTC lookup, handling returns is still a significant source of application slowdown. Reducing IBTC-related overhead by handling returns using a lower cost method is desirable.

We can eliminate the overhead of IBTC lookups for returns and just execute the return instruction directly by rewriting calls to use their fragment cache return addresses, rather than

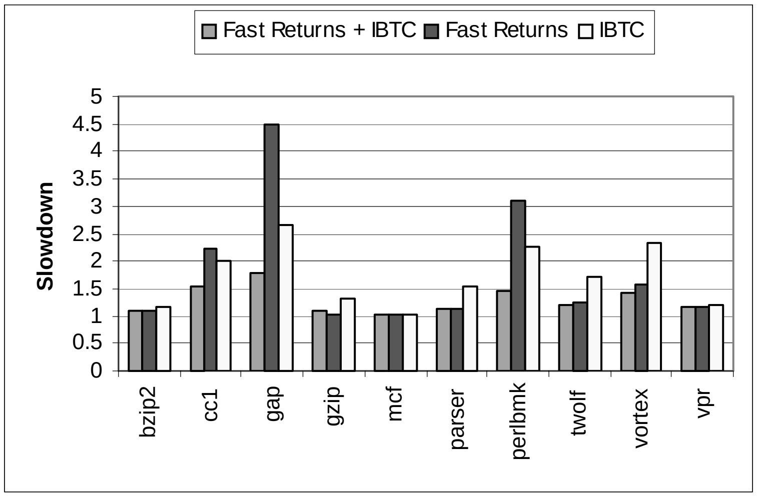

Figure 5. Strata overhead with three combinations of aggressive overhead reduction techniques.

their normal text segment return addresses. Thus, when the return executes it jumps to the proper location in the fragment cache. This technique is safe if the application does not modify the caller's return address before executing the callee's return. While it is possible to write programs that do modify the return address before executing the return, this is a violation of the SPARC ABI that compilers and assembly language programmers avoid [21].

The bar labeled "Fast Returns" in Figure 5 shows the application slowdown with fragment linking, no IBTC, and fast returns. The average slowdown across all benchmarks is about 1.8x which is slightly higher than the slowdowns obtaining using IBTC alone. The reason for this greater slowdown is that we are eliminating all return induced context switches, but context switches for other indirect branches remain. In applications where a substantial portion of the indirect control transfers are non-returns, those non-return indirect control transfers increase Strata overhead significantly.

It is possible to combine fast returns with IBTC to further reduce overhead to remedy this situation. The bar labeled "Fast Returns + IBTC" in Figure 5 shows the slowdowns using fragment linking, fast returns, and 512 entry IBTCs for non-return indirect branches. The slowdowns, averaging 1.3x, are lower than either fast returns or IBTC alone and represent our best effort so far in reducing Strata overhead. As we shall show in Section 4, these overhead reduction techniques allow us to implement two novel dynamic safety checkers whose performance is significantly better than one proposed approach to safety checking, and on par with another recently published approach. This level of performance is achieved even though both safety checkers are simple extensions to Strata.

3.3 Software Dynamic Translation Memory Overhead

In addition to execution time overhead, a program run under SDT may suffer increased memory utilization. This increased memory utilization is due to the fragment cache, the dynamic translator's data memory, and the translator's own code. Figure 6 shows the increase in maximum resident set size when an application is run with Strata's best overhead reduction techniques and a 4 megabyte fragment cache. We chose to use maximum resident set size as the measure of memory overhead since it reflects actual memory reference patterns of the executed program. For many of the benchmark applications, maximum resident set size is increased only slightly. For others, especially cc1 and twolf, the maximum resident set size is increased by more than 40 percent. In these applications, this higher overhead is due to IBTC lookup instructions in frequently executed non-return indirect branch trampolines. For modern desktop and server systems with large memories, this increase in memory utilization, while large relative to the program without Strata, is small relative to available memory. For other platforms, such as mobile devices, which have smaller available memories, this increased memory utilization may be an issue that must be addressed.

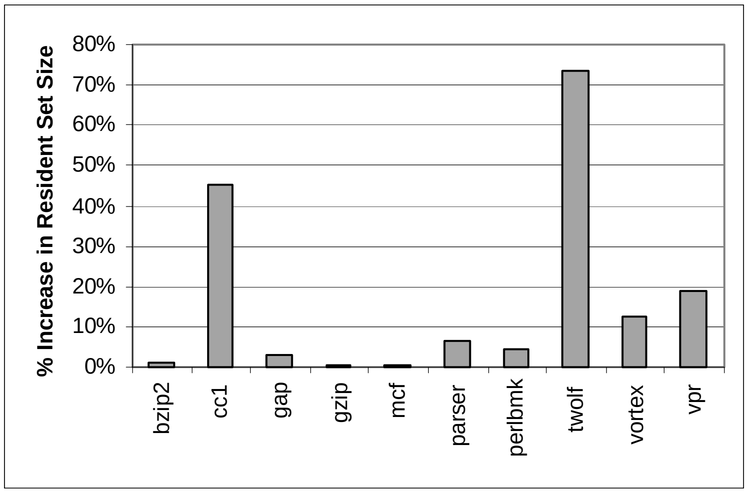

Figure 6. Increase in maximum resident set size when benchmark programs are run under Strata with a 4M fragment cache.

4 APPLICATIONS OF SOFTWARE DYNAMIC TRANSLATION

To demonstrate the effectiveness of our overhead reduction techniques in a real application domain, we used Strata to implement two different dynamic safety checkers.

4.1 Preventing Buffer-overflow attacks

A major problem in securing computer systems is the undetected presence of buffer-overflow vulnerabilities in software. Buffer overflow vulnerabilities allow malicious entities to insert nearly-arbitrary code into a program [15]. The inserted code is subsequently executed by the program, most often to ill effect. The most popular exploit for buffer overflow vulnerabilities are so-called “stack-smashing” attacks, where the buffer overflowed is allocated on the program stack. These attacks typically write a sequence of instructions onto the stack and then arrange for the address of the inserted code to overwrite a return address on the stack. This is accomplished by passing a carefully crafted source buffer to a function such as C’s `strcpy()` that does not check to see if the stack-resident destination buffer is sufficiently large to hold the source buffer.

When the attacked function returns, rather than transferring control to the caller, the malicious code is executed. The consequences of buffer overflow vulnerabilities are usually most serious when the affected program is privileged, a circumstance that allows the attacker to gain privileged access to the machine and its resources.

Implementing a Strata dynamic translator that eliminates all buffer overrun exploits without requiring access to source is trivial. We simply extend the fetch routine in the fetch/decode/translate engine (see Figure 1) to compare the PC of fetched instructions against the current stack and heap boundaries[1]. If the PC is inside the stack or heap regions, then a buffer has been overflowed and the program is on the verge of executing malicious code. At this point it is a simple matter for Strata to terminate the application or perform some other appropriate action.

A Strata-based dynamic translator that can prevent buffer overflow exploits in progress is no doubt useful. However, if the run-time overhead of such a tool is excessively high, then the applicability of this technique in real systems may be limited. Because this particular translator is such a trivial extension to Strata, and performs no code improving transformations of its own, the overhead of this technique is very nearly the overhead of the base Strata system.

Figure 7 shows the slowdown of our safety checker applications. The column labeled “Base” shows the slowdown of the base Strata with the best overhead reduction settings—512 entry IBTCs for non-return control transfers, fast returns, and fragment linking. The column labeled “Buffer” shows the slowdown of the Strata-based buffer-overflow detection tool. The slowdown for this safety checker averages 1.37x across all benchmarks. This slowdown is significantly lower than Cowan, Pu, et al’s Memguard technique, which is equivalent in power to our tool, yet

1. Locating and keeping track of stack and heap boundaries during program execution is a simple matter on many systems.

averages slowdowns of anywhere from 54x to 1743x on synthetic microbenchmarks [9]. Our safety checker compares favorably to Baratloo, Singh, et al's libverify [5]. They report slowdowns of 1.15x. However, their technique is substantially more complex than our simple extensions to Strata.

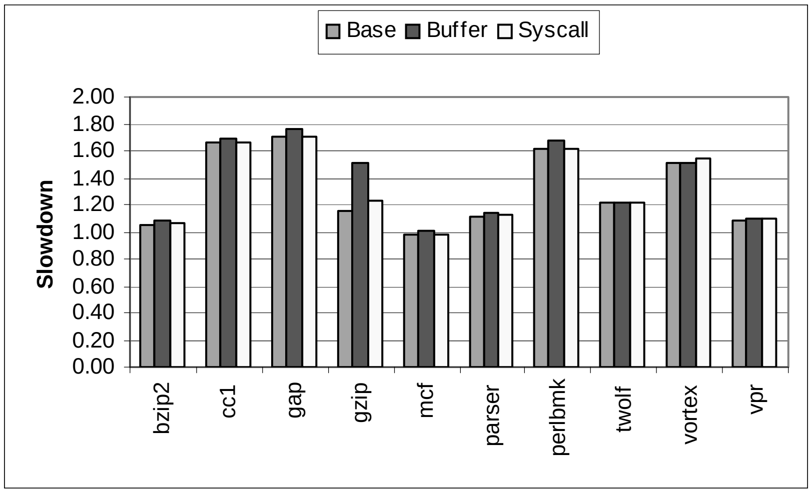

Figure 7. Slowdown of Strata-based dynamic translators. The bar labeled "Base" is the slowdown of the base Strata system with best overhead reduction. The bar labeled "Buffer" is the slowdown of the buffer-overflow monitor with best overhead reduction. The bar labeled "Syscall" is the slowdown of the system-call monitor with best overhead reduction.

4.2 Preventing unauthorized system calls

The second Strata-based safety checker that we implemented prevents unauthorized system calls in untrusted binaries. Such safety checkers are increasingly important in the era of ubiquitous networking where binary progams are downloaded from untrusted sources and executed. These binary programs may potentially contain code, malicious or otherwise, that misappropriates system resources through execution of system calls. In fact programs containing buffer overflow vulnerabilities fall into this category: most buffer-overflow exploits execute unauthorized system calls.

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(). Like the buffer overflow safety checker, policy enforcement is easily accomplished in Strata through a simple modification of the translation facility in the fetch/decode/translation engine. The translator inserts code before each system call 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.

As was the case with buffer overflow safety checker, 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 7, the column labeled "Syscall" shows the slowdown of the system call policy safety checker. The policy enforced by this safety checker is that the application may not perform an `exec()` system call. This policy is sufficiently powerful to prevent two known stack-smashing attacks on privileged Solaris programs [2, 3]. With our aggressive overhead reduction techniques in place, the slowdowns of this safety checker averages 1.32x. Again, software dynamic safety checking compares favorably in complexity and overhead to existing techniques of equivalent power.

5 RELATED WORK

Overhead is a major issue in many SDT systems and a number of overhead reduction techniques have previously been discussed in the literature. For example, the Shade simulator [8] and the Embra emulator [22] use a technique called chaining to link together cache-resident code fragments to bypass translation lookups. This technique is identical to Strata’s fragment linking and is simple but effective overhead reduction technique.

A different approach is used by the dynamic optimizer proposed by Voss and Eigenmann [20]. Their dynamic optimizer achieves low overhead on multiprocessors by running the dynamic optimizer concurrently with the application but on a different CPU. One disadvantage of their approach is the requirement of a second CPU on which to run the optimizer thread. The overhead of their approach was also evaluated using a single microbenchmark. Overheads on realistic benchmarks have yet to be published.

As in Strata, indirect branches cause difficulties for dynamic optimizers. Consequently both Dynamo [4] and DBT [oracle paper] convert indirect branches to chains of conditional branches to improve program performance. These chains of conditional branches are in a sense a simple cache for indirect branch targets. But rather than eliminate context switches as the IBTC does, the conditional branch chains remove indirect branch penalties and increase available ILP by permitting speculative execution. Since the conditional branch chains must be kept relatively short to maintain any increases in performance, an indirect branch typically terminates the conditional branch chain to handle the case when none of the conditional branch comparisons actually match the branch-target address. In the case of programs containing switch statements with large numbers of frequently executed cases, e.g., cc1 and perlbmk, the conditional branch comparisons will frequently not match the branch-target address resulting in a context switch. In Strata, the IBTC addresses this problem by accommodating a large number of indirect branch targets for each indirect branch. In our approach fewer context switches are performed, while their approach yields superior pipeline performance when the branch target is one of the few in the conditional branch chain.

Many researchers have studied the problem of buffer overrun vulnerabilities and the more general problem of restricting application resource utilization. Evans and Larochelle have proposed a technique for statically detecting many buffer overrun vulnerabilities through analysis of C source code [15]. Their approach has the obvious advantage of no run-time overhead. However, their approach does require program source. Further, static analyzers cannot detect all possible buffer overrun vulnerabilities because the problem is undecidable. The Strata buffer overflow safety checker, on the other hand, does not require program source code and prevents *all* buffer overflow exploits.

Cowan, *et al.* have proposed techniques for dynamically eliminating stack smashing attacks by detecting malicious alterations of function return addresses or by preventing those malicious alterations [9]. Their StackGuard technique can eliminate some, but not all, stack-smashing attacks through a specially modified compiler that uses a alternate procedure calling convention. In contrast, the Strata buffer overflow safety checker does not require source code and it eliminates all stack-smashing attacks. The overhead of StackGuard appears to be lower than the Strata buffer overflow safety checker. However, the overhead of both tools is low enough to be practical.

Their more powerful MemGuard technique completely eliminates stack-smashing opportunities by using OS facilities to write protect procedure return addresses during procedure activations. This approach incurs substantial overheads, often yielding orders of magnitude slowdowns in program execution. In contrast, our safety checker has significantly lower overhead with the same preventive power.

The libverify library of Baratloo, *et al.* prevents stack-smashing attacks by dynamically maintaining a stack of so-called “canary” words in parallel with the normal procedure activation stack [5]. When a procedure returns, the canary word on the parallel stack is compared with the canary word on the activation stack. If the two are different, the system concludes that a buffer overflow has occurred. The libverify library employs dynamic translation to insert the code to implement the parallel stack and canary-word comparisons at program load time. The overhead of libverify is reported to be around 15%. While the overhead of the Strata buffer overflow safety checker is slightly higher than libverify, it prevents the same class of buffer-overflow exploits. The implementation of our buffer overflow safety checker is also less complex than libverify, requiring less than 10 lines of code to implement in Strata, as opposed to libverify’s many hundreds of lines.

The Janus project proposed a sand-boxing technique that enforces a predefined system call utilization policy [14]. System calls are dynamically intercepted using an OS system call trace facility. Their system is transparent and performs sand-boxing at very low overhead. They report lower overheads than our Strata-based safety checkers. 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.

6 SUMMARY

Because software dynamic translation continues to play a significant role in modern systems, techniques that can improve the performance of SDT are very useful. In this paper we have demonstrated two such techniques that can improve the performance of SDT systems. These techniques are particularly effective in systems that perform no dynamic optimization. Our overhead reduction techniques reduce the penalties associated with indirect control transfer handling in SDT systems. One technique, the indirect branch translation cache (IBTC), allows us to reduce the cost of determining an indirect control transfer's fragment cache location by two orders of magnitude. The resulting improvement in overhead averages a factor 2.4. Another technique, fast returns, completely eliminates the overhead associated with the indirect control transfers that are used to return from function calls. When these two techniques are combined, overhead is reduced by an average factor of 3 across all benchmarks studied, and in some cases SDT overhead is completely eliminated.

Low overhead in non-optimizing SDT systems may enable and make practical the use of SDT technology in many new areas. To demonstrate this, we used our Strata SDT framework to implement two safety checkers that perform two very useful functions. The first safety checker prevents buffer-overflow attacks, and the second prevents untrusted binaries from making unauthorized system calls. Both safety checkers were implemented in Strata with a few dozen lines of code. Using our overhead reduction techniques, the safety checkers achieved performance comparable with or superior to state-of-the-art safety checkers that do not use SDT.

7 REFERENCES

[1] Plex86. http://www.plex86.org.

[2] The solaris eject buffer-overrun exploit. http://www.insecure.org/sploits/solaris.eject.html.

[3] The solaris ufsdump buffer-overrun exploit. http://www.insecure.org/sploits/Solaris.ufsdump.ufsrestore.html.

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[5] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[6] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David Gillies. Mojo: A dynamic optimization system. In *Proceedings of the ACM Workshop on Feedback-Directed and Dynamic Optimization FDDO-3*, 2000.

[7] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, Marchslash April 1998. Presented at Hot Chips IX, Stanford University, Stanford, California, August 24–26, 1997.

[8] Bob Cmelik and David Keppel. 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*, pages 128–137, May 1994.

[9] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, , and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 1998 USENIX Security Symposium*, 1998.

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

[11] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: Less is more. In *Proceedings of the 9th Internationl Conference on Architectural Support for Programming Languages and Operating Systems*, pages 202–211, November 2000.

[12] K. Ebcioglu, E. Altman, S. Sathaye, and M. Gschwind. Optimizations and oracle parallelism with dynamic translation. pages 284–295.

[13] Kemal Ebcioglu and Erik Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *24th Annual International Symposium on Computer Architecture*, pages 26–37, 1997.

[14] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 1996 USENIX Security Symposium*, 1996.

[15] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, 2001.

[16] Mendel Rosenblum. VMware's Virtual Platform: A virtual machine monitor for commodity PCs. In IEEE, editor, *Hot Chips 11: Stanford University, Stanford, California, August 15–17, 1999*, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1999. IEEE Computer Society Press.

[17] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. Vulcan: Binary translation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, April 2001.

[18] Standard Performance Evaluation Corporation. SPEC CPU2000 Benchmarks. http://www.specbench.org/osg/cpu2000.

[19] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. In *Proceedings of the ACM Workshop on Dynamic Optimization Dynamo ’00*, 2000.

[20] Michael Voss and Rudolf Eigenmann. A framework for remote dynamic program optimization. In *Proceedings of the ACM Workshop on Dynamic Optimization Dynamo ’00*, 2000.

[21] David L. Weaver and Tom Germond. *The SPARC Architecture Manual Version 9*. Prentice-Hall PTR, Upper Saddle River, NJ 07458, USA, 1994.

[22] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 68–79, May 1996.