# Overhead Reduction Techniques for Software Dynamic Translation

K. Scott[#], N. Kumar[+], B. R. Childers[+], J. W. Davidson[*], and M. L. Soffa[+]

[#]Google, Inc.
New York, New York
jks6b@virginia.edu

[+]Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
{naveen,childers,soffa}@cs.pitt.edu

[*]Dept. of Computer Science
University of Virginia
Charlottesville, VA 22904
jwd@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 systems. 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 investigate several overhead reduction techniques, including indirect branch translation caching, fast returns, and static trace formation, that can improve SDT performances significantly.*

## 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) [4], DAISY (which translates VLIW to PowerPC) [10], UQDBT (which translates IA-32 to SPARC) [20], and Transmeta's Code Morphing technology (which translates IA-32 to VLIW) [8].

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. SDT systems that perform dynamic optimization include Dynamo (which optimizes PA-RISC binaries) [1,9], Vulcan (which optimizes IA-32 binaries) [18], Mojo (which optimizes IA-32 binaries) [3], DBT (which optimizes PA-RISC binaries) [11], and Voss and Eigenmann's remote dynamic program optimization system (which optimizes SPARC binaries using a separate thread for the optimizer) [21]. 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) [7], VMware (which virtualizes either Windows or Linux) [14], and Plex86 (which virtualizes Windows for execution under Linux) [13].

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 several 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, particularly the handling of indirect control transfers and instruction cache effects. Using our measurements as a guide, we implemented 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.

## 2. Software dynamic translation

Software dynamic translation can affect an executing program by inserting new code, modifying some existing code, or controlling the execution of the program in some way. As part of the Continuous Compilation project at the University of Virginia and the University of Pittsburgh [6], we have developed a reconfigurable and retargetable SDT system [16], called Strata, which supports many SDT applications, such as dynamic optimization, safe execution of untrusted binaries [15], code decompression [17], and program profiling [12]. It is available for many platforms, including SPARC/Solaris 9, x86/Linux, MIPS/IRIX, and MIPS/Sony Playstation 2.

To realize a specific dynamic translator Strata basic services are extended to provide the desired functionality. The Strata basic services implement a 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 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. Alternatively, 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 program counter (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.

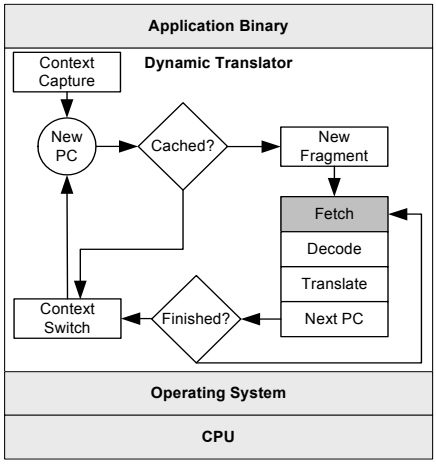



**Figure 1: Strata virtual machine**

The transfers of control from Strata to the application and from the application back to Strata are called *context switches*. On context switch into Strata via a

[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.

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 to the application occurs. As discussed below, context switches are a large part of SDT overhead.

## 3. Dynamic overhead reduction techniques

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.

### 3.1. Methodology

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 with an implementation of Strata (called "Strata-SPARC") for the Sun SPARC platform. The experiments were done on an unloaded SUN 400MHz UltraSPARC-II with 1GB of main memory. The basic Strata-SPARC dynamic 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 Strata-SPARC. We used the SPECint2K training inputs for all measurement runs.

### 3.2. Fragment linking

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.

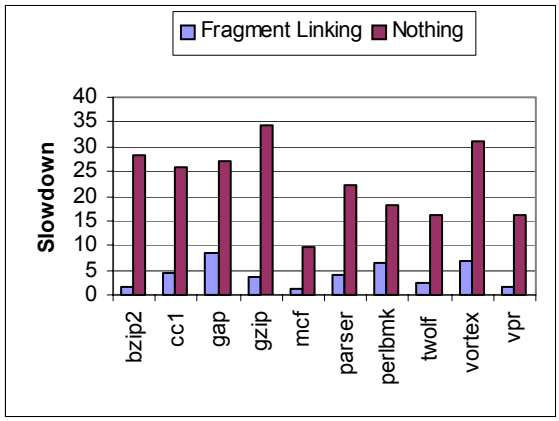



**Figure 2: Overhead reduction with fragment linking.**

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—and requires other mechanisms to lower the overhead further.

### 3.3. Indirect branch handling

The majority of the remaining overhead after applying fragment linking 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. For programs that execute large numbers

[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.

of indirect control transfer instructions, the overhead of handling the indirect branches can be substantial.

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.

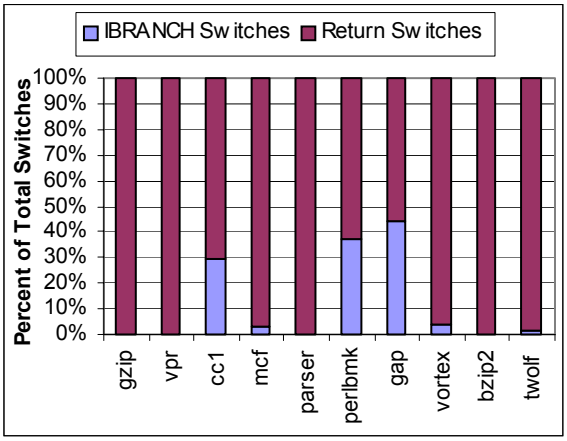
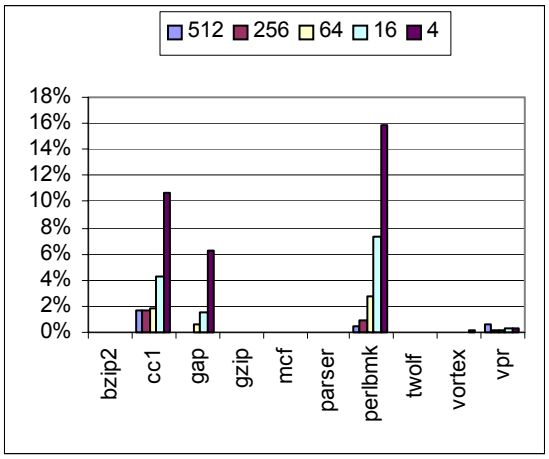



**Figure 3: Causes of context switches (with fragment linking enabled)**

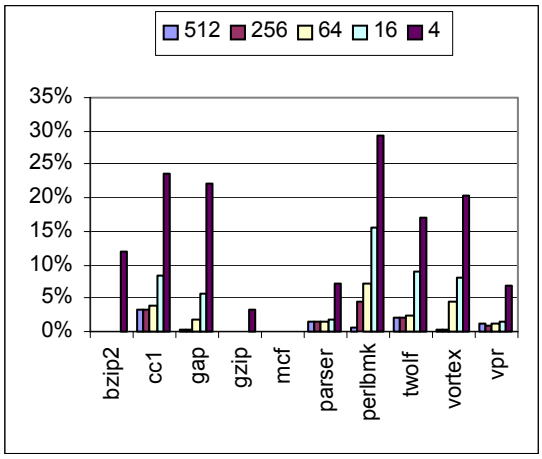
To improve Strata overhead beyond the gains achieved from fragment linking we must either find a way to reduce the latency of individual context switches to Strata, or we must reduce the overall number of switches due to indirect control transfers. The code which manages a context switch is highly-tuned,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

### 3.3.1. Indirect branch translation cache

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 simpler structure, and 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.




**(a) Miss rate with non-return indirect branches**




**(b) Miss rate with all indirect branches**
**Figure 4: IBTC miss rates.**

The inserted code saves a portion of the application context and then looks up the computed indirect branch target in the 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 tar-

get 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. Figure 4a shows miss rates when only non-return indirect control transfers are handled with IBTCs. Figure 4b 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.

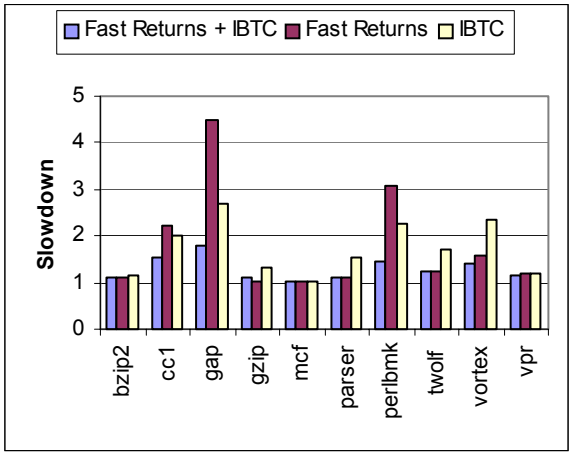



**Figure 5: Overhead Reduction with IBTC**

The performance benefits of an IBTC are substantial. In Figure 5, the white bar shows application slowdowns when using fragment linking and 512-entry IBTCs to handle indirect control transfers, including returns (the other results in Figure 5 will be discussed in Section 3.3.2). 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*.

### 3.3.2. Fast returns

Although the IBTC mechanism yields low miss rates, due to the large percentage of executed returns 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 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.

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.

## 4. Static overhead reduction techniques

While dynamic techniques can be used to tackle SDT overheads, an alternative approach can use static knowledge about a program to plan for run-time execution. This "planning approach" has less run-time analysis and code generation overhead because decisions are made off-line. In this section, we describe a first preliminary step toward using static knowledge to reduce the overhead of SDT. Similar to the dynamic overhead reduction techniques described in Figure 3, our initial approach tries to reduce the cost of context switches. It also tries to improve instruction cache locality.

Our approach uses "static plans" generated by the compiler to determine fragment code traces that reduce context switches due to indirect branches *and* improve instruction cache locality. Here, the compiler deter-

mines a "static plan" that identifies instruction traces at compile time, which can be used by SDT at run-time.

An instruction trace is a sequence of instructions on a hot path. Instruction traces improve the performance of a program by improving the hardware instruction cache locality, thereby reducing instruction cache misses. These traces can be determined by profiling a program, which can be done online as well as offline. Online profiles potentially have a high cost because dynamic instrumentation code must be used to determine instruction traces. Also, online profiles have a "lost opportunity cost", since past history must be collected to identify candidate (hot) traces.

To reduce the instrumentation and opportunity cost of finding candidate traces, the same information can be collected offline. With an offline profile, instruction traces can be identified by the compiler and preloaded into the fragment cache for subsequent execution of the program. The disadvantage of this technique is that the offline traces may not match the actual behavior of the program, particularly when the input data has a large influence on the program's execution.

Our technique uses an algorithm that we call "next heaviest edge" (NHE) to determine the traces to be preloaded into the fragment cache. NHE forms traces by starting with a seed edge from a profile that has the heaviest weight and the blocks associated with this edge are added to the trace. NHE adds new blocks to the trace by selecting the successor and the predecessor edges with the heaviest weights until an end of trace condition is encountered. The end of trace condition considers the significance of successor and predecessor edges, code duplication, and the size of a trace.

The NHE algorithm forms instruction traces across indirect branches. The algorithm identifies such control transfers when forming traces and predicts that an indirect transfer will "stay on trace". Because the exact target address of an indirect branch is not known until runtime, checking code is emitted in the trace at each indirect branch. This check verifies that the target address of an indirect branch is indeed the next subsequent block on the trace. If the block is not on the trace, a context switch is made into Strata to handle the indirect transfer. Our preliminary implementation does not attempt to reduce the number of context switches due to trace mispredictions (i.e., when going off the trace). However, in practice, an indirect branch is likely to have multiple possible targets, which means an inordinate number of context switches may occur into Strata.

To investigate the overhead reduction with static traces, we profiled several SPEC2K benchmarks with the training data set. The profile was used to determine instruction traces with NHE. These traces were saved in a file that is preloaded whenever Strata-SPARC is invoked. In the subsequent run, we used the reference input set from SPEC2K for each benchmark.

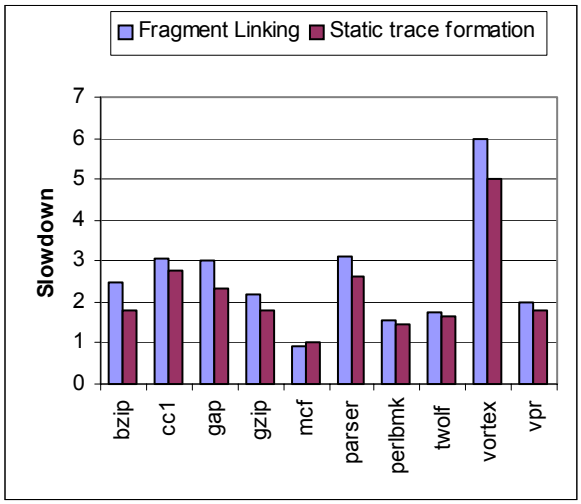



**Figure 6: Performance improvement with static trace formation**

Figure 6 shows the slowdown of preloading traces with Strata over not preloading the traces with Strata. The numbers in this graph were run on a Sun Blade 100 with a 500 MHz UltraSPARC IIi, 256 MB memory, and gcc 3.1 with optimization level -O3. The overhead numbers for fragment linking are different in this graph than in Figure 2 due to the different machine platform.

Figure 6 shows that static trace formation improves performance by reducing the number of context switches due to indirect transfers and instruction cache misses. The performance improvement over fragment linking ranges from 0% to 39%, with an average of 15%. From our experiments, the improvement is due to *both* a reduction in the number of context switches and instruction cache misses. However, the improvement is not as large as using the IBTC and fast returns (see Section 3.3). The improvement with static trace formation is influenced by the prediction accuracy of the statically formed traces. In the current scheme, the accuracy is moderate, with most traces being exited early.

One way to improve the current scheme is to combine it with an IBTC and Fast Returns. In this case, when the run-time check on the indirect branch finds that the transfer is off trace, the IBTC can be consulted to find the target address without a context switch into Strata. We are implementing this scheme and expect it to do better than the IBTC and fast returns alone because the scheme also addresses instruction cache locality. Our initial results, however, are encouraging because they show that static information can be effectively used to reduce the overhead of SDT.

## 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 [4,8,10,20], emulation of operating systems (e.g., VMWare, Plex86), and machine simulation [7,22]. While most of these systems have been built for a single purpose, there has been recent work on general infrastructures for SDT that are similar to Strata.

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. 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 [19].

Another flexible framework for SDT is DynamoRIO [2], which is a library and set of API calls for building SDTs 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 [7] and the Embra [22] 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 [21]. One of the major sources of overhead in a system like Strata are indirect branches.

Consequently both Dynamo [1] and Daisy [10] 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.

## 6. Summary

Reducing the overhead of software dynamic translation (SDT) is critical for making SDT systems practical for use in production environments. Using the SPECint2K benchmarks, we performed detailed measurements to determine major sources of SDT overhead. Our measurements revealed that the major source of SDT overhead comes from handling conditional and indirect branches. For example, conditional branches, when handled naively, can result in slowdowns as much 34 times (average 22x) over a directly executed binary.

Guided by our measurements, we developed techniques to reduce these overheads. One technique, called fragment linking, reduces overhead caused by conditional branches by rewriting the trampoline code to transfer control directly to the appropriate fragment rather than doing a context switch. This technique reduces SDT overhead by a factor of 5 (22x to 4x).

Our measurements also showed that indirect branches were a significant source of overhead. To reduce the number of context switches caused by indirect branches, we used an indirect branch translation cache. This cache maps indirect branch-target addresses to their fragment cache location. With a small 512-entry cache, the overall slowdown was further reduced from an average 4.1x to an average of 1.7x.

To reduce overheads further, we developed a technique to better handle indirect branches that were generated because of return statements. For function returns where the fragment cache holds the return address, function returns can be rewritten to return

directly to the fragment cache return address thereby avoiding a context switch. This technique reduced the SDT slowdown to an average of 1.3x.

Finally, we investigated the usefulness of determining instruction traces statically and using this information to reduce the number of context switches and improving instruction cache locality. This technique resulted in a performance improvement of up to 39% (average 15%) over fragment linking. Our preliminary results demonstrate that static information can be successfully used to guide SDT and reduce its overhead.

While overheads in the range 2 to 30 percent (with no other optimizations applied) may be acceptable for some applications, for other applications even a modest slowdown is unacceptable. We are continuing to develop other techniques for reducing SDT overhead. Preliminary results indicate that by applying the techniques described here along with some dataflow analysis of the executable, it may be possible to eliminate SDT overhead entirely. If achieved, this would make SDT a powerful tool for helping software developers achieve a variety of important goals including better security, portability, and better performance.

## 7. Acknowledgements

This work was supported in part by National Science Foundation, Next Generation Software, grants CNS–0305198, CNS–0305144, CNS–0203945 and CNS–0203956.

## 8. References

[1] V. Bala, E. Duesterwald, and S. Banerjia, “Dynamo: A transparent dynamic optimization system”, *ACM Conf. on Programming Language Design and Implementation*, pp. 1–12, 2000.

[2] D. Bruening, T. Garnett, and S. Amarasinghe, “An infrastructure for adaptive dynamic optimization”, *Int’l. Symp. on Code Generation and Optimization*, March 2003.

[3] W-K Chen, S. Lerner, R. Chaiken, and D. Gillies, “Mojo: A dynamic optimization system”, *Workshop on Feedback-Directed and Dynamic Optimization*, 2000.

[4] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates, “FX!32: A profile-directed binary translator”, *IEEE Micro*, 18(2), pp. 56–64, April 1998.

[5] C. Cifuentes, B. Lewis, and D. Ung, “Walkabout—A retargetable dynamic binary translation framework”, *Workshop on Binary Translation*, 2002.

[6] B. Childers, J. Davidson, and M. L. Soffa, “Continuous compilation: A new approach to aggressive and adaptive code transformation”, *NSF Next Generation Software Workshop*, during the *Int’l. Parallel and Distributed Processing Symposium*, April 2003.

[7] B. Cmelik and D. Keppel, “Shade: A fast instruction-set simulator for execution profiling”, *ACM SIGMETRICS Conf. on the Measurement and Modeling of Computer Systems*, pp. 128–137, 1994.

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

[9] E. Duesterwald and V. Bala, “Software profiling for hot path prediction: Less is more”, *Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 202–211, November 2000.

[10] K. Ebcioglu and E. Altman, “DAISY: Dynamic compilation for 100% architecture compatibility”, *24th Int’l. Symp. on Computer Architecture*, pp 26–37, 1997.

[11] K. Ebcioglu, E. Altman, S. Sathaye, and M. Gschwind, “Optimizations and oracle parallelism with dynamic translation”, *Int’l. Symp. on Microarchitecture*, pp. 284–295, 1999.

[12] N. Kumar and B. Childers, “Flexible instrumentation for software dynamic translation”, *Workshop on Exploring the Trace Space*, during the *Int’l. Conference on Supercomputing*, 2003.

[13] Plex86, http://www.plex86.org

[14] M. Rosenblum, “Virtual platform: A virtual machine monitor for commodity PCs”, *Hot Chips 11*, 1999.

[15] K. Scott and J. Davidson, “Safe virtual execution using software dynamic translation”, In *Annual Computer Security Application Conference*, 2002.

[16] K. Scott, N. Kumar, S. Velusamy, B. R. Childers, J. W. Davidson, and M. L. Soffa, “Retargetable and reconfigurable software dynamic translation”, *Int’l. Symp. on Code Generation and Optimization*, March 2003.

[17] S. Shogan and B. Childers, “Compact binaries with code compression in a software dynamic translator”, *Design Automation and Test in Europe*, 2004.

[18] A. Srivastava, A. Edwards, and H. Vo, “Vulcan: Binary translation in a distributed environment”,Technical Report MSR–TR–2001–50, Microsoft Research, 2001.

[19] J. Troger and J. Gough, “Fast dynamic bianry translation—The Yirr-Ma framework”, *In. Proc. of the 2002 Workshop on Binary Translation*, 2002.

[20] D. Ung and C. Cifuentes, “Machine-adaptable dynamic binary translation”, *Proc. of the ACM Workshop on Dynamic Optimization*, 2000.

[21] M. Voss and R. Eigenmann, “A framework for remote dynamic program optimization”, *Proc. of the ACM Workshop on Dynamic Optimization*, 2000.

[22] E. Witchel and M. Rosenblum, “Embra: Fast and flexible machine simulation”, Proc. of the ACM SIGMETRICS Int’l. Conf. on Measurement and Modeling of Computer Systems, pp. 68–79, 1996.