Evaluating Fragment Construction Policies for SDT Systems

Jason D. Hiser, Daniel Williams,
Adrian Filipi, Jack W. Davidson
Department of Computer Science
University of Virginia
{hiser,dww4s,adrian,jwd}@cs.virginia.edu

Bruce R. Childers
Department of Computer Science
University of Pittsburgh
childers@cs.pitt.edu

Abstract

Software Dynamic Translation (SDT) systems have been used for program instrumentation, dynamic optimization, security policy enforcement, intrusion detection, and many other uses. To be widely applicable, the overhead (runtime, memory usage, and power consumption) should be as low as possible. For instance, if an SDT system is protecting a web server against possible attacks, but causes 30% slowdown, a company may need 30% more machines to handle the web traffic they expect. Consequently, the causes of SDT overhead should be studied rigorously.

This work evaluates many alternative policies for the creation of fragments within the Strata SDT framework. In particular, we examine the effects of ending translation at conditional branches; ending translation at unconditional branches; whether to use partial inlining for call instructions; whether to build the target of calls immediately or lazily; whether to align branch targets; and how to place code to transition back to the dynamic translator. We find that effective translation strategies are vital to program performance, improving performance from as much as 28% overhead, to as little as 3% overhead on average for the SPEC CPU2000 benchmark suite. We further demonstrate that these translation strategies are effective across several platforms, including Sun SPARC UltraSparc IIi, AMD Athlon Opteron, and Intel Pentium IV processors.

Categories and Subject Descriptors B.m [**Hardware**]: Miscellaneous, C.4 [**Performance of Systems**]: Performance attributes, D.3.0 [**Programming Languages**]: General, D.3.4 [**Programming Languages**]: Processors - *Code Generation, Compilers, Incremental Compilers, Interpreters, Optimization, Run-time Environments*

General Terms Algorithms, Measurement, Performance, Design, Experimentation.

Keywords Software Dynamic Translator, Low Overhead, Performance, Dynamic Translation Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VEE'06 June 14–16, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-332-6/06/0006…$5.00.

1. Introduction

Over the years software dynamic translation (SDT)—the programmatic modification of a running program's binary instructions—has become an increasingly useful technique in the system implementor's repertoire. A wide variety of systems can be classified as software dynamic translators, including dynamic optimizers, dynamic binary translators, dynamic instrumentation systems, dynamic software updaters, and certain emulators and simulators.

SDT affords system designers unprecedented flexibility in controlling and modifying a program's execution. This flexibility allows software dynamic translation to be used to accomplish several objectives not easily achieved via other means. For instance, SDT may be used to overcome the barriers to entry associated with the introduction of a new OS or CPU architecture. Transmeta's Code Morphing technology is used for this very purpose; i.e., allowing unmodified Intel IA-32 binaries to run on the low-power, VLIW Crusoe processor [7]. Similarly, the UQDBT system dynamically translates Intel IA-32 binaries to run on SPARC-based processors [23], and FX!32 dynamically translates x86 binaries to run on Alpha processors [5].

In addition to allowing designers to overcome cost barriers to new platform acceptance, the flexibility of SDT has proven useful for other purposes. For instance, Shade uses SDT to implement high-performance instruction set simulators [6]. Embra uses SDT to implement a high-performance operating system emulator [24]. Dynamo and Mojo use SDT to improve the performance of native binaries [1, 4], and DAISY uses software dynamic translation to evaluate the performance of novel VLIW architectures and accompanying optimization techniques [9]. The Kerninst [22, 21] and Vulcan [20] systems use SDT to insert program monitoring instrumentation into running programs. More recently SDT has been used to ensure safe execution of untrusted binaries [11, 14, 15, 17].

Despite the many applications of SDT and the lively state of research into novel uses of SDT, one major obstacle for more pervasive use of dynamic translators is the cost of applying transformations at runtime. When an SDT system adds too much overhead, in the form of execution time, memory requirements, disk requirements, or network traffic, the system is less likely to be used. For instance, an SDT system may be used to protect a web server from malicious attackers trying to gain access to the system. However, if the SDT system causes the web server to run 30% slower than without the SDT system, a large web-hosting firm may need up to 30% more CPUs to handle the same amount of traffic during periods of peak demand. This increase corresponds to a 30% increase

in initial hardware costs, but also up to a 30% increase in peak electricity usage, a 30% increase in space required to store the devices, and up to a 30% increase in peak cooling costs for the building! With such significantly increased cost, the SDT system may be an undesirable solution. Instead, if the SDT system introduces only 1% overhead, it may well be worth the extra hardware and utility costs to prevent malicious attacks. Consequently, we see that reducing SDT overhead is of utmost importance for the pervasive use of SDT technology.

The causes of overhead in SDT systems need to be thoroughly and rigorously evaluated. A fundamental operation done by an SDT system is to translate and cache code fragments (the basic code granularity of translation). To this end, we examine a variety of fragment construction policies, their possible benefits and drawbacks. In particular, we study and evaluate some of the design decisions that were made within Strata [14], a publicly available, retargetable SDT system supporting three instruction sets: IA-32, SPARC, and MIPS.

We find that several of Strata's initial design decisions were inefficient and caused overhead in several ways. Some decisions caused extra instructions to be executed, while other decisions hurt the performance of the processor and memory hierarchy, including branch prediction accuracy and cache locality.

The major contributions of this paper are:

- It describes several decisions that must be made when building a dynamic translation system for low overhead.
- The paper empirically determines the best set of fragment creation policies to use when building a low overhead SDT system.
- The paper demonstrates significant overhead reduction on three machines, reducing overhead to as little as 3% on average for an AMD Athlon Opteron 244.
- This work presents evidence that the fragment creation techniques described are efficient at reducing overhead across a variety of architectures, not just a single architecture used for initial experimentation.
- This work explains the remaining causes of overhead that we were unable to remove.
- Finally, it puts forth the notion that the causes of overhead can be due to inhibiting the underlying hardware from peak performance rather than due to increases in instruction counts or time spent within the SDT system.

The remainder of this paper is organized as follows: Section 2 describes Strata, the SDT system used in this work. Section 3 presents our findings using experiments designed to find the lowest overhead option of many different translations strategies. Section 4 highlights other research to reduce SDT overhead, and finally Section 5 summarizes our findings.

2. Strata

Strata is an SDT system designed for high retargetability and low overhead translation. Strata has been used for a variety of applications including system call monitoring, dynamic download of code from a server, and enforcing security policies [12, 25]. This section describes some of the basic features of Strata which are important to understanding the experiments presented later. For an in depth discussion of Strata, please refer to previous publications [14, 16, 17].

2.1 Overview

Strata operates as a co-routine with the program binary it is translating, as shown in Figure 1. As the figure shows, each time Strata encounters a new instruction address (i.e., PC), it first checks to see if the address has been translated into the *fragment cache*. The fragment cache is a software instruction cache that stores portions of code that have been translated from the native binary. The fragment cache is made up of *fragments*, which are the basic unit of translation. If Strata finds that a requested PC has not been previously translated, Strata allocates a fragment and begins translation. Once a termination condition is met, Strata emits any *trampolines* that are necessary. Trampolines are pieces of code emitted into the fragment cache to transfer control back to Strata. Most control transfer instructions (CTIs) are initially linked to trampolines (unless its target previously exists in the fragment cache). Once a CTI's target instruction becomes available in the fragment cache, the CTI is linked directly to the destination, avoiding future uses of the trampoline. This mechanism is called *Fragment Linking* and avoids significant overhead associated with returning to Strata after every fragment [14].

Strata's translation process can be overriden to implement a new SDT use. The basic Strata system includes several default behaviors that control the creation of code fragments. These default decisions can be changed based on a particular SDT use. This paper evaluates which choices are indeed the best as the default.

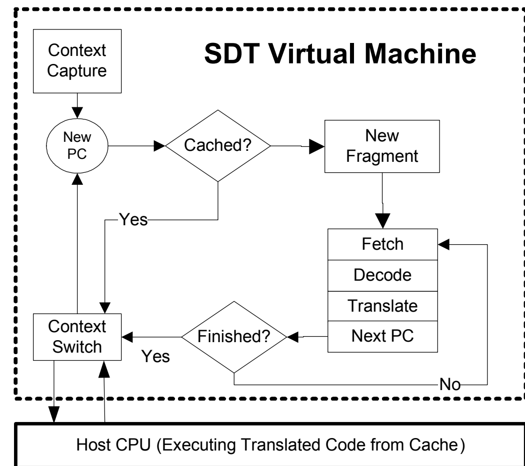

Figure 1: High-level overview of how Strata operates.

2.2 Indirect Branches

Indirect branches[1] require special handling. Since indirect branch targets can change during each execution of the instruction, they cannot be linked at translation-time to their target in the fragment cache. To handle indirect branch targets efficiently, Strata emits a sequence of instructions that perform a table lookup. The table lookup maps application PC's to their fragment PC equivalents. On

1. For this work, indirect calls are considered a special type of indirect branch.

a table miss, control is transferred back to Strata which builds the requested fragment if necessary, updates the lookup table and finally transfers control to the destination fragment.

Return instructions are a special kind of indirect branch that Strata optimizes via a method called *Fast Returns*. Instead of emitting the normal sequence of instructions to do a table lookup, it directly emits the return instruction. Normally this would cause the translator to lose control of the running application after the first return instruction is executed, since control would transfer back to an application PC. Strata prevents this by translating call instructions specially. Call instructions normally write their return address into a specified location. Strata translates these instructions to write the corresponding fragment cache return address. Such translation causes the return instruction to return to another fragment within the fragment cache, and Strata can consequently maintain control. We choose to enable this option as it provides for an aggressive baseline in which to attempt to remove more overhead. Please see previous publications for further detail about indirect branch handling mechanisms [16].

3. Fragment Translation Policies

This section describes the translation strategies used within Strata, and experimental evidence for changing some of Strata's initial design decisions. Section 3.1 first describes the setup used for these experiments.

3.1 Experimental Setup

All performance experiments in Sections 3.2–3.7 are performed on an AMD Athlon Opteron 244 (1.8GHz) running Fedora Core 4, with Linux kernel 2.6.11. The kernel and all software is built using the IA-32 instruction set. Benchmarks are compiled with the GNU gcc compiler and the options -O3 -fomit-frame-pointer with dynamic linking. The entire set of SPEC CPU2000 benchmarks are used for performance analysis. Some graphs presented use the PAPI statistic collection mechanisms to gather information about processor events such as instruction cache misses via hardware counters [2].[1] Graphs are normalized to native execution time when possible. In these graphs, smaller numbers represent better performance. The fragment cache size is set to 255MB, so that removing blocks from the fragment cache is never necessary. However, only 1 benchmark exceeded 10MB of fragments in the fragment cache with the optimized configuration.

3.2 Normal Instructions

Normal instructions are simple instructions such as add or mov that cause no control transfer. Such instructions are copied as-is from the application text into the fragment cache and no special handling is needed. Consequently, translation of these instructions leaves little chance that overhead is created, and no design alternatives to evaluate.

3.3 Direct, Conditional CTIs

Direct, conditional CTIs, such as beq L1 or jecxz L2, require translating the target address from an application address to a fragment cache address. They are also good points to consider ending fragment construction. To understand why, consider that CTIs sometimes have a highly-biased outcome, they are either taken or not taken for the entire run of the program. Branches that detect error conditions are often of this form. If the translator were to continue translating code at either the fall-through or target of the branch, then the translator is performing extra work and creating extra pressure in the fragment cache and hardware instruction cache. On the other hand, if the translator stops translating at the branch and it is not highly biased, extra instructions will be needed along the fall-through path to link the branch's fall-through to the fragment for not taken code.

It is possible to conditionally stop the fragment build 1) if the target of the branch already exists in the fragment cache; 2) if the fall-through of the branch already exists in the fragment cache; 3) if both the fall-through and the target exist in the fragment cache; or 4) if either the target or the fall-through exist in the fragment cache.

Figure 2 shows the results when each of these possibilities is implemented within Strata.[2] The results indicate that always ending the fragment at a conditional branch generally has a significant performance penalty, especially in the integer benchmarks which are less memory bound and more control-intensive. Performance is generally improved over Strata's baseline (always stop) by conditionally ending the fragment, but the best performance comes from unconditionally continuing to build the fragment. Over 5% improvement can be seen on average. The only other fragment building mechanism that comes close is ending if both the fall-through and the target previously exist in the fragment cache. This mechanism's performance is close to unconditionally continuing because the stopping condition is rarely met.

Consequently, we believe that never ending a fragment at a conditional branch is the best way to reduce overhead. All further experiments will not end fragment building at conditional branches.

3.4 Handling Direct, Unconditional CTIs

Unconditional, direct CTIs, such as jmp L1, present an opportunity for the translator to generate better code by eliminating the jump instruction and continuing translation at the destination of the jump, or *eliding* the instruction. Sometimes eliminating the CTI helps program performance by reducing instruction count. In other cases, eliminating jumps causes code to be duplicated in the fragment cache, which in turn impacts instruction cache performance.

Figure 3 compares performance when Strata eliminates jump instructions with emitting unconditional jump instructions into the fragment cache (when the target fragment already exists in the fragment cache). The figure shows that some benchmarks benefit slightly from the reduction in instruction count of eliminating the branch instructions. Other benchmarks, however, have a dramatic degradation due to excessive code duplication. On average, keeping the jump instructions yield a 0.5% improvement. Thus, we con-

1. These graphs, due to software integration issues, are collected from a Intel Pentium IV Xeon machine, running Red Hat Linux 7.3 2.96-110, with kernel version 2.6.11perfctr-2.6.

2. Reported SPEC results represent three runs of SPEC. *Average, int ave*, and *fp ave* are the geometric mean of the SPEC numbers reported for all benchmarks, integer benchmark, and floating-point benchmarks, respectively. Performance results may vary from previous publications due to bug fixes, upgrades to machines and different operating systems and architectures.

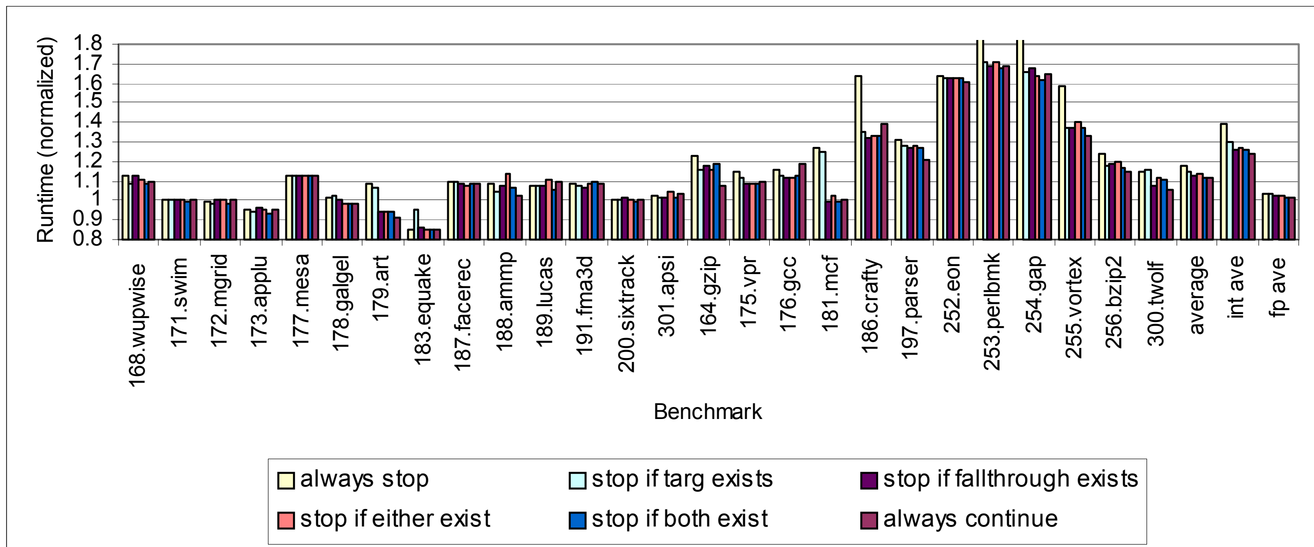

Figure 2: Performance of Strata as the fragment termination condition of conditional CTIs is changed.

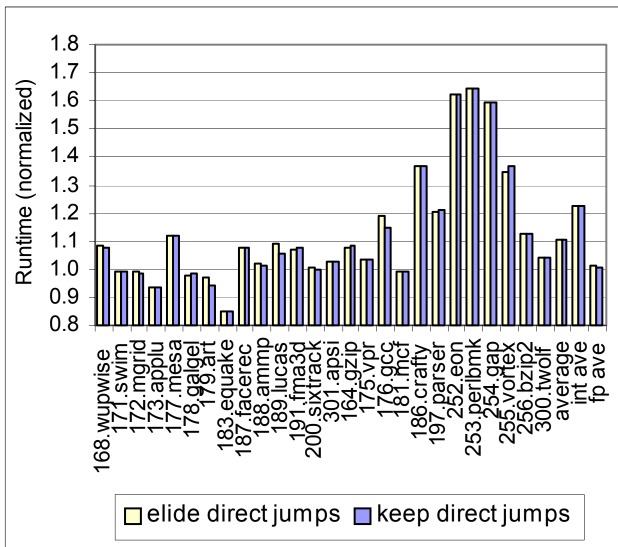

Figure 3: Performance of Strata as the fragment termination condition of unconditional CTIs is changed.

clude that keeping the jump instructions is wise, and will use that policy for all experiments in the remainder of the paper.

3.5 Direct, Call Instructions

Direct call instructions, while not as common as direct branches, are still an integral part of most programs. Programs written in object-oriented languages promote the use of the direct calls to invoke the many small functions often found in these types of programs. Consequently, it is important for a dynamic translator to handle direct call instructions efficiently. Sections 3.5.1–3.5.2 discuss alternatives when translating direct call instructions.

3.5.1 Partial Inlining

Strata (and other SDT systems) often implement partial inlining. Partial inlining refers to translating a call instruction into a write of the return address (a push instruction in an IA-32 instruction set) and continuing instruction translation at the destination of the call instruction. Partial inlining was believed to help performance by eliminating unnecessary CTIs and reducing instruction count.

Figure 4 shows the performance of applications run under Strata with and without partial inlining enabled. As the figure shows, partial inlining is actually a loss for most programs. In fact, the performance loss is over 14% on average for the integer benchmarks, and over 30% on two benchmarks! We believe that the primary reason for the loss of performance is that the branch predictor's return address stack is not yielding proper predictions because call instructions are eliminated when partial inlining is used. There are also negative affects from code duplication. Figure 5 shows that partial inlining results in more branch mispredictions. On average, with partial inlining, there are 2.47 times more mispredictions than native execution, while there are only 1.24 times more mispredictions without partial inlining.

Although partial inlining has significant performance penalties by negatively impacting the branch predictor, it may yield more opportunities for dynamic optimization. As baseline Strata performs no dynamic optimizations, it is impossible to overcome the performance loss of partial inlining. However, any work striving to gain performance based on the increased context of partial inlining must first overcome the negative impacts of using partial inlining before performance gains can be realized.

For our work, we conclude that not using partial inlining is the best translation policy, and all remaining experiments do not use it.

3.5.2 Lazy Translation of Call Targets

Since the target of a direct call is known at translation time, it is possible to *speculatively translate* the target into the fragment cache when the call site is translated. Speculatively translating the

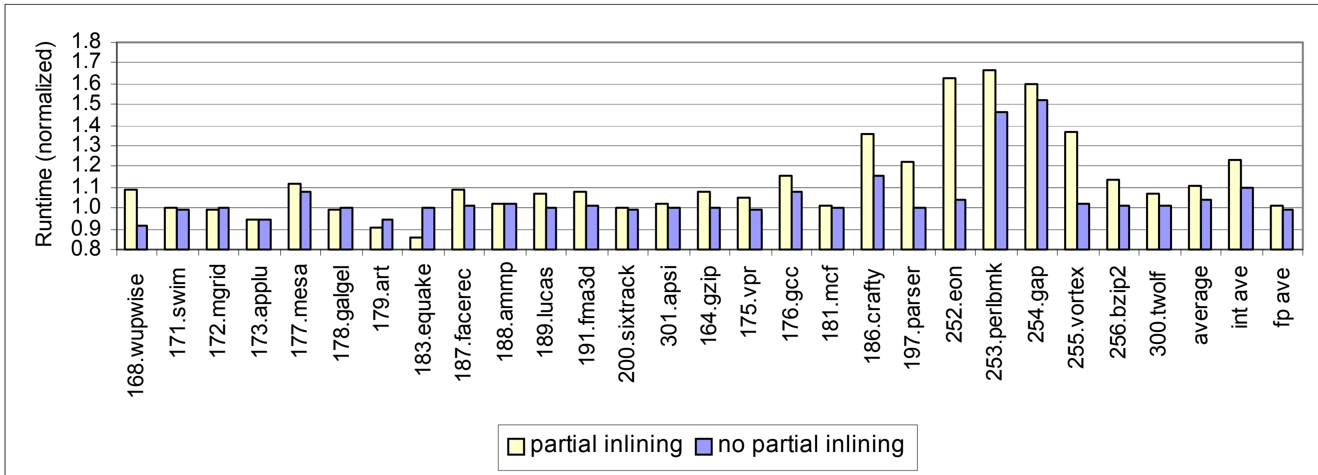

Figure 4: Performance of Strata when partial inlining is used compared to no partial inlining.

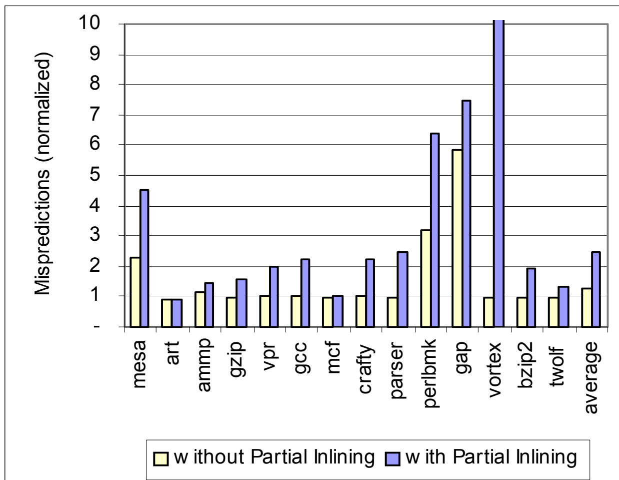

Figure 5: Branch mispredictions with and without partial inlining. (Missing benchmarks are a result of third party tools failing to integrate properly with system tools.)

target can potentially save a context switch back to the SDT system. Unfortunately, speculative translation can also increase pressure on the fragment and instruction caches and waste translation time. An alternative is *lazy call translation* that translates the call target once the call instruction actually executes. Although the final code that is materialized is the same, the placement of the code can be dramatically different. Figure 6 shows how speculative and lazy call translation affect the performance of applications run with Strata.

The figure shows that translating calls speculatively generally yields little performance gain. In fact, several benchmarks (*lucas, art,* and *perlbmk*) have significantly worse performance. We believe this is due to decreased instruction cache locality. To confirm this belief, examine results from PAPI in Figure 7. The figure shows the reduction in the instruction cache misses when calls are lazily instead of speculatively translated. The figure indicates that some benchmarks have dramatically reduced instruction cache miss rates. On average, over 14% of instruction cache misses are eliminated. Unfortunately, the reduction in misses does not always translate to increased performance, as some benchmarks are not limited by instruction cache fetch rates.

Thus, we believe that call target should be lazily translated, and all remaining experiments use lazy translation for call targets.

3.6 Fragment Alignment

Fragments are almost always the destination of a branch instruction. By forcing the fragment to start on a cache line boundary, or *aligning* the fragment, the processor can fetch more useful instructions the cycle after fetching a branch that is predicted taken. However, forcing the alignment of fragments also leaves a useless *hole* in the fragment cache between the two fragments. These holes potentially cause extra conflicts in the instruction cache and increased pressure in the fragment cache.

To better understand the tradeoffs between using fragment alignment and not using it, consider Figure 8. The figure shows the performance with and without fragment alignment. The benchmarks generally show an improvement with fragment alignment, although some have no gain or even a performance loss. The most telling indicator that the fragment alignment is beneficial is that the integer benchmarks have nearly 2% improvement. These programs are also the most control-intensive, where alignment is most helpful. Although the overall benefit it slight, we believe that fragment alignment is generally good for performance and should be used. Subsequent experiments use fragment alignment.

3.7 Trampoline Placement

A *trampoline* is a portion of code emitted into the fragment cache that helps perform a context switch and transfer control from the application into the translator. Trampolines are typically small, on the order of 10 instructions. However, any CTI whose destination is not already in the fragment cache needs a unique trampoline. Consequently, a single fragment with 10 branches may need 10 trampolines. Initially Strata placed the trampolines for a fragment

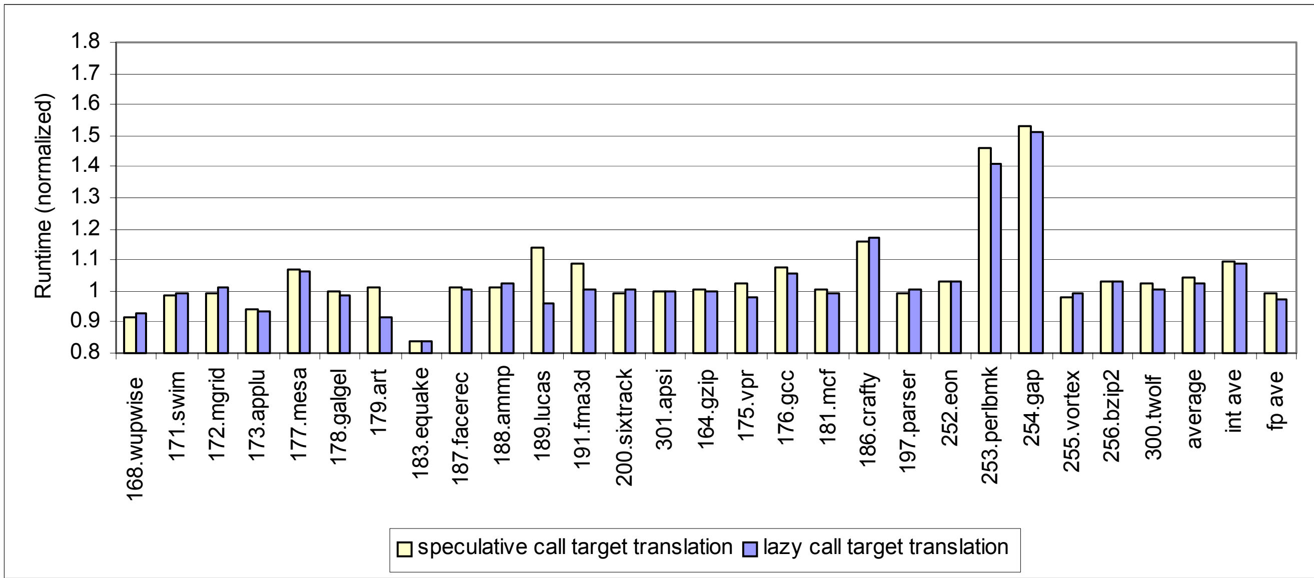

Figure 6: Results comparing lazy call target translation versus speculative translation.

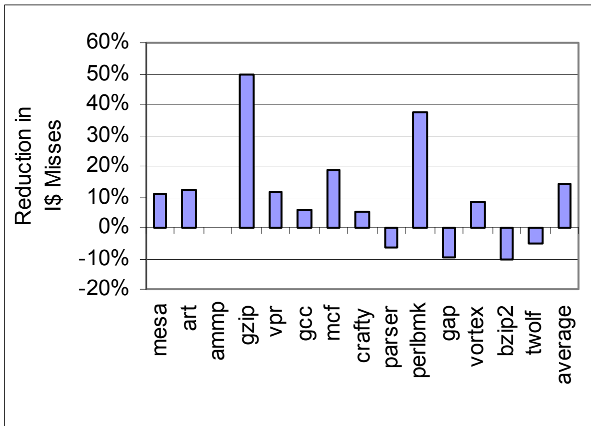

Figure 7: Reduction in instruction cache misses when call targets are translated lazily.

into the fragment cache immediately after the fragment. However, since most trampolines are used only once (once a trampoline is used, Strata translates the requested destination, and fixes direct control transfer instructions to branch directly to their intended fragment destination), it may be desirable to move the trampoline into a separate area of the fragment cache. Moving trampolines to their own area keeps the frequently used application code closer together in the fragment cache. This can potentially reduce the conflicts in the instruction cache.

Figure 9 shows the results of placing trampolines in a separate area of the fragment cache, called the "trampoline pool". The first bar shows the performance when a fragment's trampolines are placed immediately after a fragment. The second bar shows the performance when trampolines are placed in the pool. As the figure shows, most applications differ little, however, the applications with higher instruction cache pressure do show some improvement. Figure 10 confirms this by plotting the reduction in instruction cache misses when a trampoline pool is used. In fact, over 18% of the cache misses are eliminated by moving trampolines to their own area. Consequently, we believe that keeping fragment trampolines in a pool is worthwhile, and we use this policy for the remainder of the experiments.

3.8 Cross Platform Verification

All previously described experiments were run on an AMD Athlon Opteron 244. However, the construction strategies used to maximize performance for this machine work well for other machines. In particular, finite-sized instruction caches and hardware return address stacks are common features of most desktop and server machines. Unfortunately, fully exploring all possible design combinations on a wide variety of architectures is prohibitively expensive. Instead, we examine the performance of Strata's initial configuration compared to the configuration optimized for the AMD machine. Figure 11 and Figure 12 show results for an Ultrasparc IIi[1] and an Intel Pentium IV Xeon[2]. The first bar in each figure shows the application performance for Strata's initial configuration: End construction at a conditional CTI, always elide unconditional CTIs, always use partial inlining, speculative translation of call instruction targets, do not align branch targets, and keep fragment trampolines with the fragment. The second bar shows the application's performance with the revised configuration: Never end construction on a conditional CTI, emit unconditional CTI's when possible, avoid partial inlining, translate call

1. The SUNSpro cc compiler is used with optimization level -fast -xO5. Dynamic linking is enabled
2. The GNU gcc compiler is used with optimization level -O3 -fomit-frame-pointer. Dynamic linking is enabled

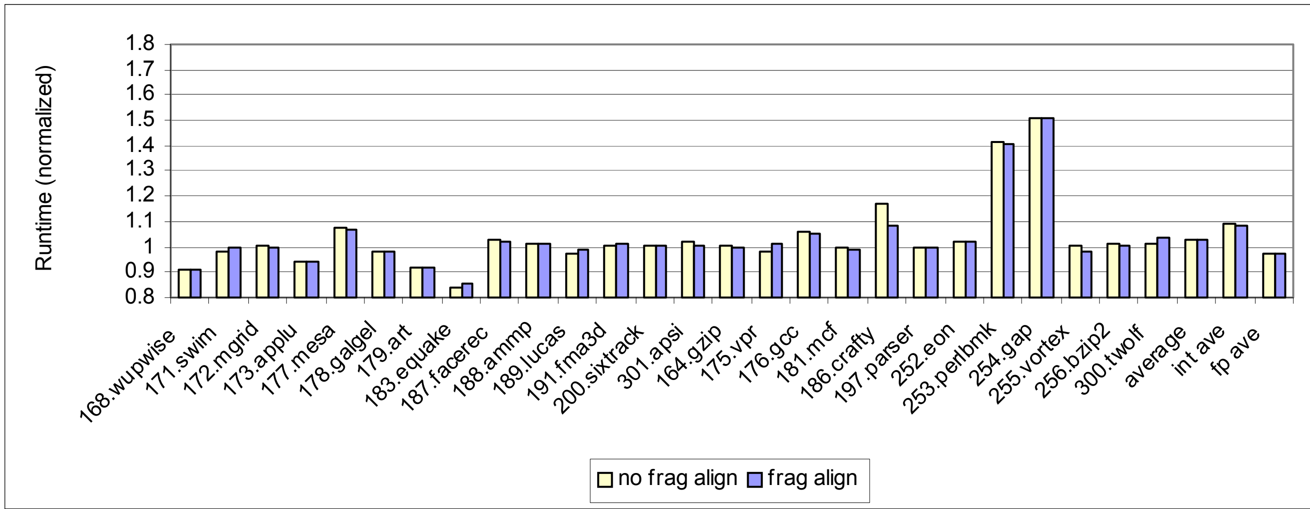

Figure 8: Performance of applications under Strata when fragment alignment is used.

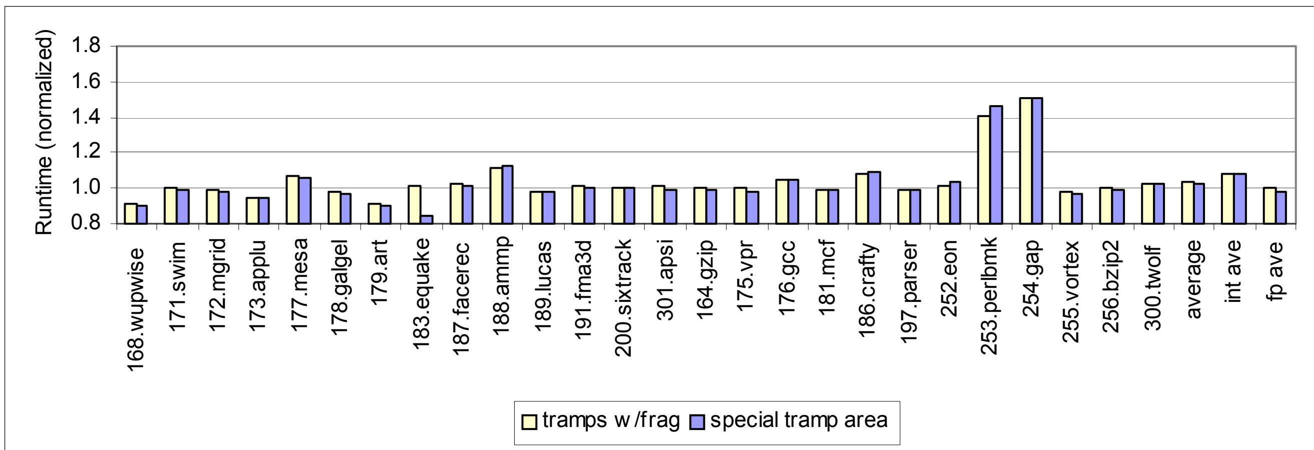

Figure 9: Performance of applications under Strata with fragment trampolines are adjacent to fragments versus a separate trampoline pool.

sites lazily, align fragments on instruction cache line boundaries, and place trampolines in a pool.

The figure shows that the optimized fragment building techniques greatly improve the performance of both machines; thus, our techniques are sound across a variety of architectures. Furthermore, machines with deeper pipelines and higher memory latencies (compared to CPU cycle times) will benefit more from these proposed translation strategies. This is particularly true because branch mispredictions and cache misses have a higher performance penalty in this case.

3.9 Remaining Causes of Overhead

By choosing the fragment building strategies outlined in previous sections, the overhead that Strata introduces to native applications has been greatly reduced: From 16% to 3% for an Opteron, 13% to 4% for an UltraSparc, and 28% to 11% for a Pentium IV Xeon. In fact, many of the programs have little or no overhead. Some programs stand out as an exception, namely *gcc, crafty, perlbmk, mesa,* and *gap*. We believe the overhead for these benchmarks are from three primary causes: Extra instructions executed, extra instruction cache misses, and extra branch mispredictions.

3.9.1 Extra Instructions

Figure 13 shows the instruction count of the benchmarks when run under Strata with optimized fragment building on a Pentium IV Xeon. As the figure shows, the benchmarks that have no overhead also do execute extra instructions. In these benchmarks, most of the time is spent executing within the fragment cache and the translation time is minimal. The remaining benchmarks execute 5–20% more instructions. With the optimized fragment building mechanisms, Strata only adds extra instructions to the fragment cache when translating indirect branches or other infrequent special cases. Unfortunately, as shown in Figure 14, the poor performing benchmarks all have a significant number of (non-return) indirect branches executed. Furthermore, the two figures together show the

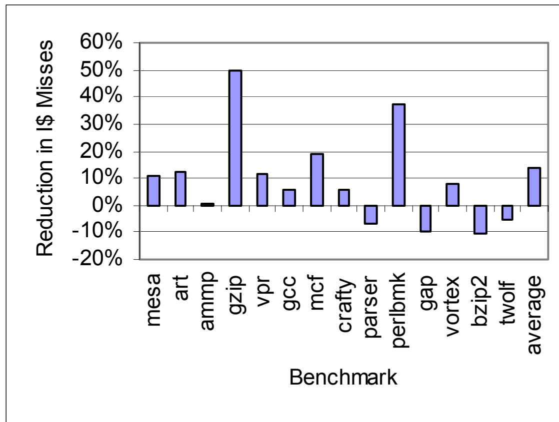

Figure 10: Percent reduction in instruction cache misses with trampoline pool.

number of indirect branches is related to the number of extra instructions, providing strong evidence that nearly all extra instructions executed come from handling indirect branches.

3.9.2 Hardware Interactions

Although it is interesting to note that the extra instructions executed are related to indirect branch handling, it is also important to understand why the execution time of benchmarks with significant overhead is not proportional to the number of indirect branches executed per instruction. We believe the disproportionately high overhead (especially on the Pentium IV machine) of *mesa, gcc, perlbmk* and *gap* are related to extra instruction cache pressure and extra branch mispredictions. Figure 15 provides evidence to support this theory. The graph plots:

$$\frac{\text{instruction cache misses(native)}}{\text{instructions(native)}} - \frac{\text{instruction cache misses(Strata)}}{\text{instructions(Strata)}}$$

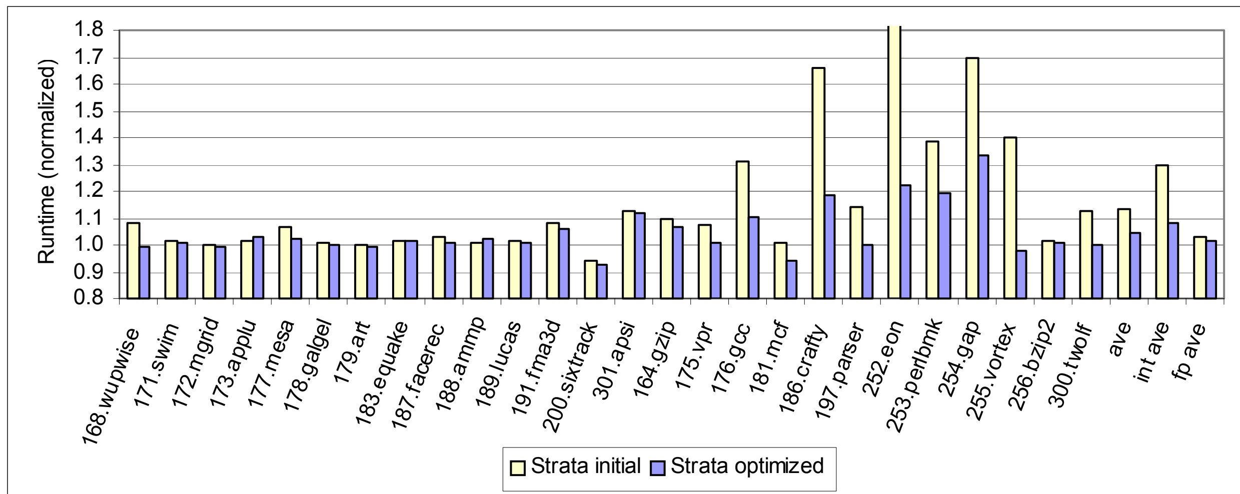

Figure 11: Performance on Strata with initial configuration and optimized configuration for UltraSparc IIi.

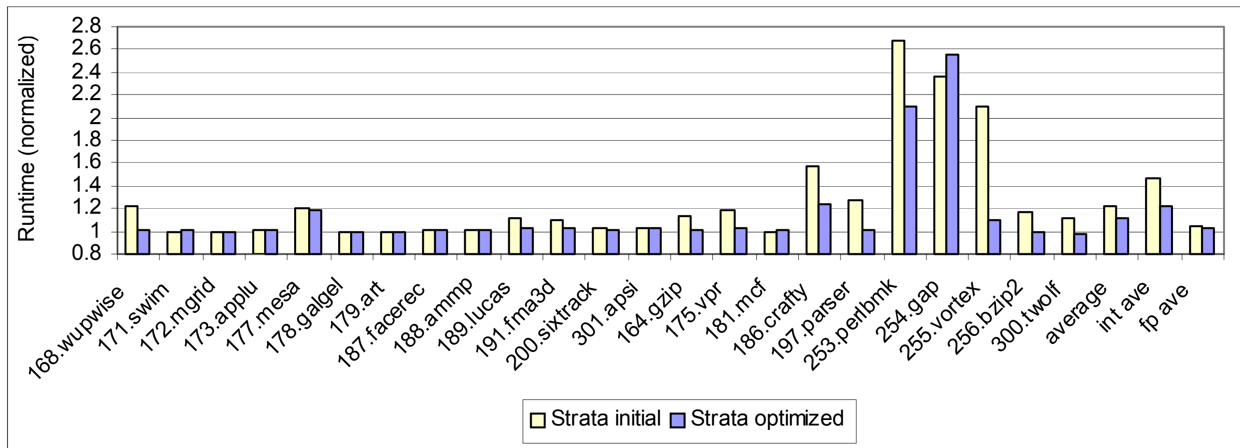

Figure 12: Performance of Strata with initial configuration and optimized configuration for Pentium IV.

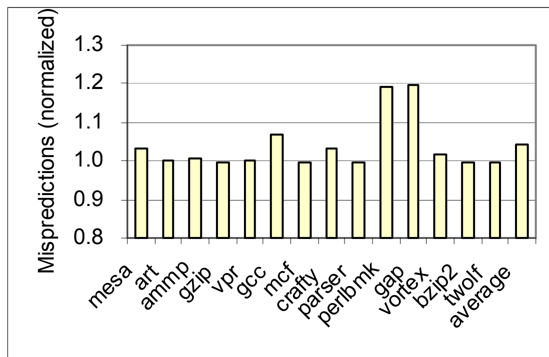

Figure 13: Instruction count of benchmarks with Strata, normalized to native execution.

In essence, this formula calculates the difference in instruction cache misses per instruction between native execution and Strata execution. The figure also plots the difference in branch mispredictions per instruction and data cache misses per instruction. The well-behaved benchmarks not only have no extra instructions, they also show very few extra cache misses or mispredictions per instruction. The benchmarks with higher overhead show a different pattern. Instead, they show significant increases in cache misses and branch mispredictions, accounting for the significant extra overhead seen in these benchmarks. The poorest performing benchmark, *gap*, has the greatest increase in branch misprediction rates. This poor branch predictor performance is because the majority of call instructions in *gap* are indirect calls, resulting in extremely poor use of the hardware return address stack when Strata is performing dynamic translation.

Also, in deeper pipelines and as pressure on caches increases, these effects are more pronounced. This is evidenced by the fact that the overhead is higher for the 30-stage pipeline of the Pentium IV than for the 15-stage pipeline in the AMD Athlon Opteron.

4. Related Work

Smith and Nair describe *dynamic basic blocks*, which are the most straightforward method of handling code cache layout [18]. With dynamic basic blocks, the dynamic translator continues code layout until it reaches a branch instruction. At that point a trampoline is inserted that returns control of the application back to the dynamic translator. Strata's original fragment layout scheme behaved similarly to dynamic basic block creation [14, 17]. However, it would terminate a fragment after every conditional or indirect branch, but continue decoding unconditional branches and calls, *eliding* them. HDTrans performs code layout slightly differently [19]. They elide unconditional branches, but they continue translating the fall through instructions after direct calls. This creates basic blocks that more closely match original layout of the program text.

More advanced tracing is implemented in DynamoRIO [3] and Pin [13]. DynamoRIO creates a sharp divide between its basic block cache and its trace cache. DynamoRIO selects traces to optimize-based on next executing tail, or NET [8]. Under this scheme, once a trace head becomes hot, the trace is created by following the execution of the path tail, assuming that the path taken is a frequently executed path. Alternately, Pin builds traces without first profiling them as frequently executed. Pin ends traces on unconditional branches, or if it has translated too many conditional branches or too many total instructions. Recently, Pin has been extended to use the *last-executed iteration* (LEI) to better select traces for optimization [10]. Table 1 summarizes how different translators end fragments.

All of these code cache layout implementations have intuitive arguments for their benefits, however each must translate and execute fragments (at least until hot paths have been selected for optimization). Unfortunately, no previous studies have quantified the effects of the initial fragment construction techniques have on SDT performance. The work presented here differs from previous work in that it attempts to quantify and select the best fragment construction methods for overhead reduction.

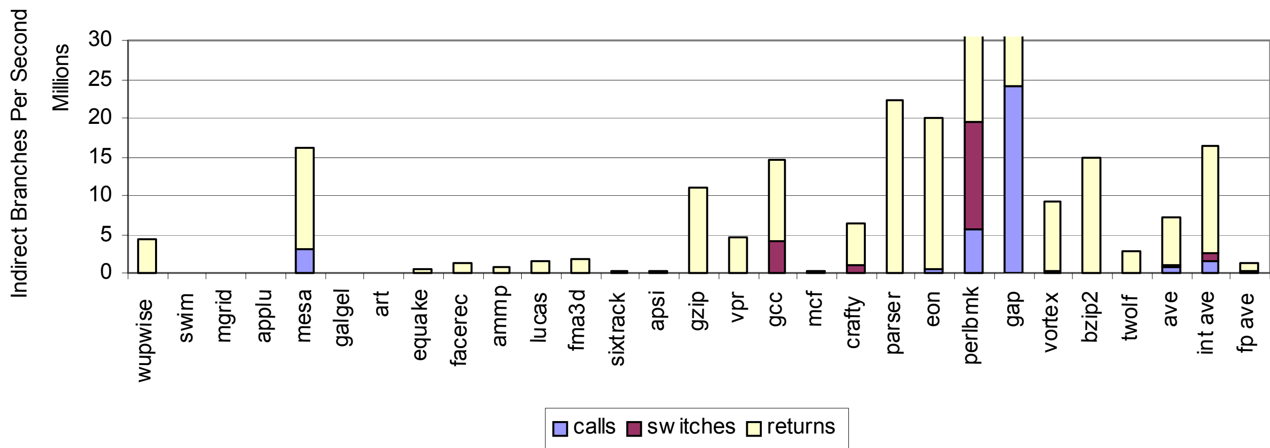

Figure 14: Indirect branches executed per second under native execution on AMD Athlon.

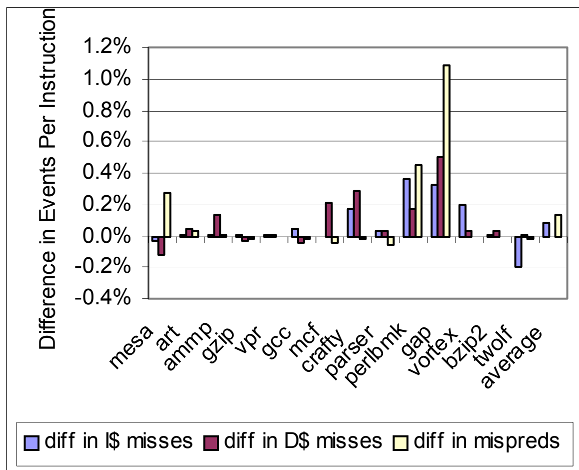

Figure 15: Difference in instruction cache misses and branch mispredictions per instruction between native execution and execution under Strata.

Table 1: Summary fragment termination conditions in different dynamic translators.

Translator	Uncond. CTI Handling	Cond. CTI Handling	Direct Call Handling	Max Insns
Original Strata	Always elide	Always end	Always inline	Never end
Opt. Strata	Elide if not exists	Always continue	Never inline	4096
HDTrans	Elide if not exists	End if targ exists	Never inline	
DynamoRIO	Always elide	Always end	Always inline	End if max insns
Pin	Always elide	Always continue[a]	Always inline	End if max insns

a. Pin terminates a trace after a “pre-defined number of conditional control transfers” [13].

5. Conclusions

Software dynamic translation (SDT) has been used extensively in the past and has a wide variety of interesting opportunities for future use. In order for SDT to be more pervasively used, SDT systems must add as little extra processor time, memory usage, disk usage and network traffic as possible. This paper has examined methods for fragment creation in SDT systems and evaluated their effectiveness within the Strata SDT framework. We have found that fragment creation should not end at conditional branches, and should end only at unconditional branches when the target fragment already exists within the fragment cache. We have further found that translating call instructions using a partial inlining technique negatively affects the hardware branch predictor and lazily translating call site targets improves instruction cache locality. Instruction cache locality is also improved by placing fragment trampolines into a special area of the fragment cache that is disjoint from the application’s instructions. In addition to evaluating the fragment building techniques on an AMD Athlon Opteron 244, the paper further validates the findings on two other machines, an Intel Pentium IV Xeon and a Sun UltraSparc IIi. We find that overhead can be reduced from 16%, 13%, and 28% to 3%, 4%, and 11% for these Opteron, UltraSparc, and Xeon, respectively on the entire SPEC2000 benchmark suite. Lastly, we examine the remaining causes of overhead and find that it is directly related to the handling of indirect branches, partially due to executing extra instructions, but also due to decreased hardware performance on the transformed program.

6. Acknowledgements

This research was support by by DARPA under agreement number FA8750-04-2-0246 and the National Science Foundation under grants CNS–0305144, CNS–0524432, and CNS–0305198. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] BALA, V., DUESTERWALD, E., AND BANERJIA, S. Dynamo: A transparent dynamic optimization system. In *PLDI ’00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (New York, NY, USA, June 2000), ACM Press, pp. 1–12.

[2] BROWNE, S., DONGARRA, J., GARNER, N., LONDON, K., AND MUCCI, P. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Supercomputing ’00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)* (Washington, DC, USA, 2000), IEEE Computer Society, p. 42.

[3] BRUENING, D., DUESTERWALD, E., AND AMARASINGHE, S. Design and implementation of a dynamic optimization framework for windows. In *Proceedings of the ACM Workshop on Feedback-Directed and Dynamic Optimization FDDO-4* (December 2001).

[4] CHEN, W.-K., LERNER, S., CHAIKEN, R., AND GILLIES, D. Mojo: A dynamic optimization system. In *Proceedings of the ACM Workshop on Feedback-Directed and Dynamic Optimization FDDO-3* (December 2000).

[5] CHERNOFF, A., HERDEG, M., HOOKWAY, R., REEVE, C., RUBIN, N., TYE, T., YADAVALLI, S. B., AND YATES, J. FX!32: A profile-directed binary translator. *IEEE Micro 18*, 2 (Mar. 1998), 56–64. Presented at Hot Chips IX, Stanford University, Stanford, California, August 24–26, 1997.

[6] CMELIK, B., AND KEPPEL, D. Shade: A fast instruction-set simulator for execution profiling. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, May 1994), ACM Press, pp. 128–137.

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

[8] DUESTERWALD, E., AND BALA, V. Software profiling for hot path prediction: less is more. In *ASPLOS-IX: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2000), ACM Press, pp. 202–211.

[9] EBCIOGLU, K., AND ALTMAN, E. DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer architecture* (New York, NY, USA, 1997), ACM Press, pp. 26–37.

[10] HINIKER, D., HAZELWOOD, K., AND SMITH, M. D. Improving region selection in dynamic optimization systems. In *MICRO 38: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 141–154.

[11] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. Secure execution via program shepherding. In *11th USENIX Security Symposium* (August 2002).

[12] KUMAR, N., MISURDA, J., CHILDERS, B. R., AND SOFFA, M. L. Instrumentation in software dynamic translators for self-managed systems. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed systems* (New York, NY, USA, 2004), ACM Press, pp. 90–94.

[13] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), ACM Press, pp. 190–200.

[14] SCOTT, K., AND DAVIDSON, J. Strata: A software dynamic translation infrastructure. In *IEEE Workshop on Binary Translation* (September 2001).

[15] SCOTT, K., AND DAVIDSON, J. W. Safe virtual execution using software dynamic translation. In *Proceedings of the 18th Annual Computer Security Applications Conference* (Las Vegas, NV, December 2002), pp. 209–218.

[16] SCOTT, K., KUMAR, N., CHILDERS, B., DAVIDSON, J. W., AND SOFFA, M. L. Overhead reduction techniques for software dynamic translation. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium* (2004), IEEE Computer Society, p. 200.

[17] SCOTT, K., KUMAR, N., VELUSAMY, S., CHILDERS, B., DAVIDSON, J. W., AND SOFFA, M. L. Retargetable and reconfigurable software dynamic translation. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 36–47.

[18] SMITH, J., AND NAIR, R. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

[19] SRIDHAR, S., SHAPIRO, J. S., AND BUNGALE, P. P. Hdtrans: A low-overhead dynamic translator. In *Proceedings of the 2005 Workshop on Binary Instrumentation and Applications* (September 2005), IEEE Computer Society.

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

[21] TAMCHES, A., AND MILLER, B. P. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the 3rd Symposium on Operating Systans Design and Implementation (OSDI-99)* (Berkeley, CA, Feb. 22–25 1999), Usenix Association, pp. 117–130.

[22] TAMCHES, A., AND MILLER, B. P. Using dynamic kernel instrumentation for kernel and application tuning. *The International Journal of High Performance Computing Applications 13*, 3 (Fall 1999), 263–276.

[23] UNG, D., AND CIFUENTES, C. Machine-adaptable dynamic binary translation. In *Proceedings of the ACM Workshop on Dynamic Optimization Dynamo '00* (2000).

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

[25] ZHOU, S., CHILDERS, B. R., AND SOFFA, M. L. Planning for code buffer management in distributed virtual execution environments. In *VEE '05: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (New York, NY, USA, 2005), ACM Press, pp. 100–109.