Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems

JASON D. HISER, DANIEL W. WILLIAMS, WEI HU, JACK W. DAVIDSON,
and JASON MARS, University of Virginia
BRUCE R. CHILDERS, University of Pittsburgh

Software Dynamic Translation (SDT) is used for instrumentation, optimization, security, and many other uses. A major source of SDT overhead is the execution of code to translate an indirect branch's target address into the translated destination block's address.

This article discusses sources of Indirect Branch (IB) overhead in SDT systems and evaluates techniques for overhead reduction. Measurements using SPEC CPU2000 show that the appropriate choice and configuration of IB translation mechanisms can significantly reduce the overhead. Further, cross-architecture evaluation of these mechanisms reveals that the most efficient implementation and configuration can be highly dependent on the architecture implementation.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement—*Documentation*; H.4.0 [**Information Systems Applications**]: General

General Terms: Experimentation, Measurement, Performance

Additional Key Words and Phrases: Fast returns, IBTC, indirect branch, indirect jump, return cache, sieve, software dynamic translation

ACM Reference Format:
Hiser, J. D., Williams, D. W., Hu, W., Davidson, J. W., Mars, J. and Childers, B. R. 2011. Evaluating indirect branch handling mechanisms in software dynamic translation systems. ACM Trans. Architec. Code Optim. 8, 2, Article 9 (July 2011), 28 pages.
DOI = 10.1145/1970386.1970390 http://doi.acm.org/10.1145/1970386.1970390

9

1. INTRODUCTION

Software Dynamic Translation (SDT) is a technology that enables software malleability and adaptivity at the instruction level by providing facilities for runtime monitoring and code modification. Many useful systems have been built that apply SDT,

---

This article extends the authors' previous work entitled "Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems" published in *Proceedings of the International Symposium on Code Generation and Optimization (CGO'07)*.

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-07-2-0029 and the National Science Foundation under grants CNS-0305144, CNS-0305198, CNS-0551492, CNS-0509115, CNS-0305198, and CNS-0551560. 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 Air Force Research Laboratory or the U.S. Government.

Authors' addresses: J. D. Hiser (corresponding author), D. W. Williams, W. Hu, J. W. Davidson, and J. Mars, Computer Science Department, University of Virginia, 151 Engineer's Way, Charlottesville, VA 22904; email: hiser@virginia.edu; B. R. Childers, University of Pittsburgh, Pittsburgh, PA.

DOI 10.1145/1970386.1970390 http://doi.acm.org/10.1145/1970386.1970390

including optimizers, security checkers, binary instruction set translators, and program instrumenters. For example, in Apple Computer's transition from a PowerPC platform to an Intel platform, they use a software dynamic translator. This translator, called Rosetta, converts PowerPC instructions into IA-32 instructions and optimizes them [Apple Computers 2006; Transitive Corporation Ltd. 2006]. The translator is integrated directly into the operating system, making the conversion transparent to the user. Other binary translators include Transmeta's code morphing system that translates IA-32 instructions to VLIW instructions [Ditzel 2000], UQDBT that dynamically translates Intel IA-32 binaries to run on SPARC processors [Ung and Cifuentes 2000], DAISY that translates PowerPC instructions to VLIW instructions [Ebcioğlu and Altman 1997; Ebcioğlu et al. 2001], and a variety of others [Zheng and Thompson 2000; Baraz et al. 2003; Gschwind et al. 2000; Chernoff et al. 1998]. Computer architecture tools like Shade and Embra use SDT to implement high-performance simulators [Cmelik and Keppel 1994], while Mojo and Dynamo dynamically optimize native binaries to improve performance [Bala et al. 2000; Chen et al. 2000]. SDT has been used to ensure the safe execution of untrusted binaries [Kiriansky et al. 2002; Scott and Davidson 2001b; 2001a; Scott et al. 2003; Hu et al. 2009].

Despite many compelling SDT applications, a sometimes critical drawback of the technology is the execution overhead incurred when running an application under the control of an SDT system. The mediation of program execution adds overhead, possibly in the form of time, memory size, disk space, or network traffic. For an SDT system to be viable, its overhead must be low enough that the cost is worth the benefit. For example, a SDT system might be used to protect critical server applications. If the protection system overhead is high, total ownership costs will be increased (e.g., the number of servers necessary for a desired throughput rate will be increased to offset the overhead). If the protection system imposes only a small overhead, say a few percent or less, then it is more likely to be used. Consequently, it is vital that SDT overhead be minimal if the technology is to be widely applied.

A major source of SDT overhead stems from the handling of Indirect Branches (IBs). Consider the graphs in Figure 1 which show the overhead (normalized to native execution) of a high-quality SDT system with naïve IB translation on an Opteron 244 processor, and the number of IBs per second executed by each benchmark. Naïve IB translation means that the SDT system regains control during each IB and performs the steps necessary to emulate the IB. Inspection of the graphs shows that there is a strong correlation between the IB execution rate and the overhead incurred by the SDT system; applications with high IB execution rates incur high SDT overhead. For example, `253.perlbmk` shows that with 19.4 million IBs executed per second, that the SDT system runs 39.4 times slower than native execution. Our results (presented in Section 5) show that efficient handling of IBs can improve performance to as little as 3% slower than native execution.

To address this problem, this work evaluates methods for efficiently handling IBs in SDT systems. This article makes the following contributions.

—Comprehensive evidence of the importance of efficiently handling IBs on different processor architectures.
—A thorough analysis of different IB translation mechanisms, including data cache handling, instruction cache handling, and a mixed method, on three popular processors.
—Algorithmic descriptions and example implementations of proposed techniques for handling IBs.
—A direct comparison between several mechanisms to exploit the regularity of return-type indirect branches.

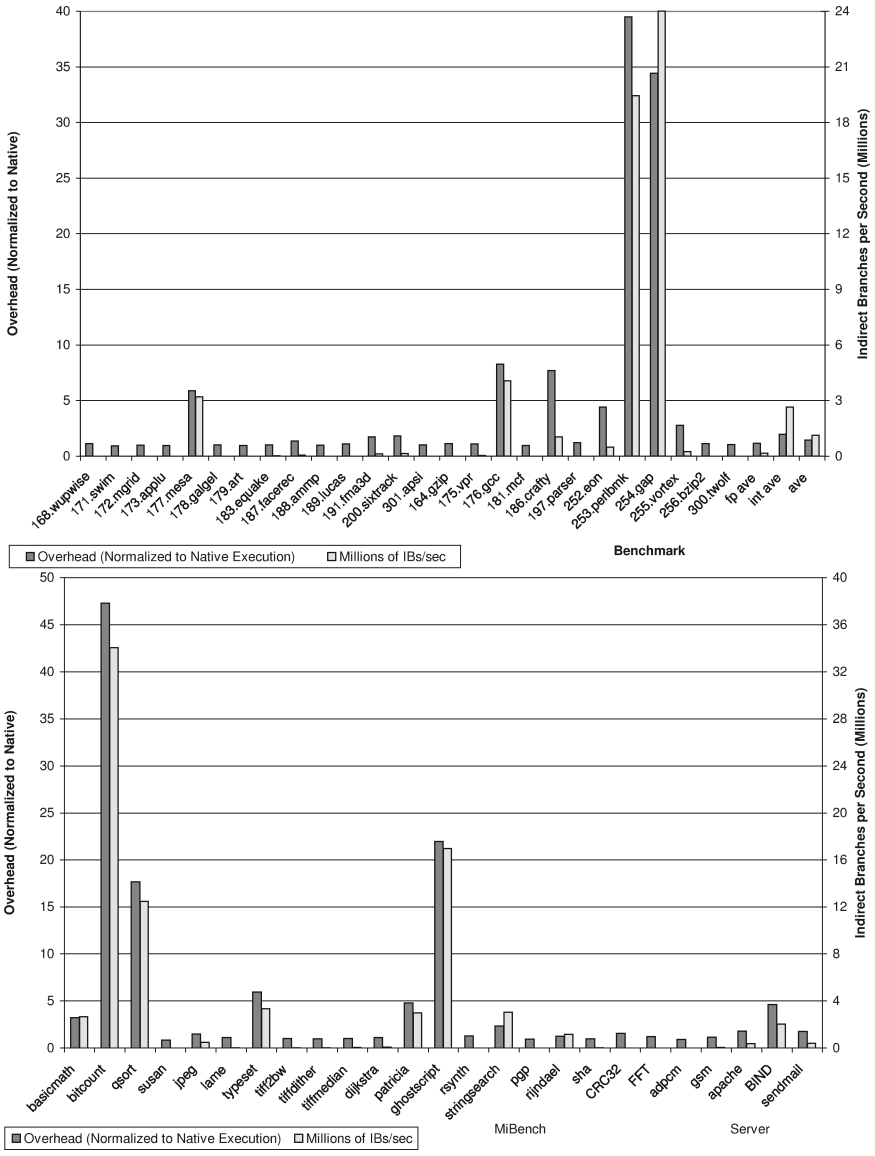


Fig. 1. Overhead of SDT with naïve IB translation (normalized to native execution) and IB execution rate (in millions of indirect branches per section). Overhead is calculated by dividing the execution time with SDT to the execution time without SDT. For example, `253.perlbmk` executes nearly 40 times slower than native execution and has 20 million IBs per second.

—A novel improvement on standard inline cache entires which gains as much as 10% execution time improvement on some benchmarks, and up to 36% improvement on `254.gap` from the SPEC benchmark suite.
—Experimental evidence on three architectures (Sparc ULTRASparc-IIi, Intel Pentium 4 Xeon, and AMD Opteron 244) that the best method for handling IBs depends on the features of the target architecture such as addressing modes, branch predictors, cache sizes, and the ability to efficiently preserve architecture state.

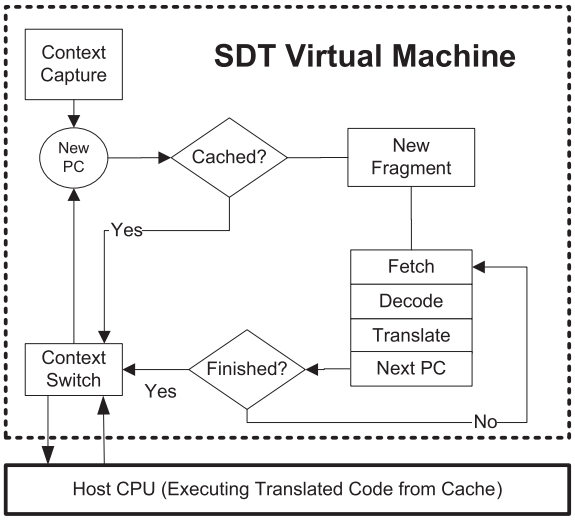


Fig. 2. Strata virtual machine translating and executing application instructions.

A key finding from our evaluation is the observation that no single method for handling IBs is always the best across architectures and programs. The best method for an architecture/program is highly dependent on the underlying processor capabilities.

The remainder of this article is organized as follows. Section 2 gives a brief overview of SDT and Section 3 has a detailed description of the IB handling mechanisms evaluated in this article. Section 4 describes the experimental framework in which the IB handling mechanisms are evaluated and Section 5 presents our findings. Sections 6 and 7 discuss related work and summarize our conclusions.

2. SOFTWARE DYNAMIC TRANSLATION OVERVIEW

This section describes some of the basic features of dynamic translation systems which are important for understanding the experiments presented later. For an in-depth discussion of these systems, please refer to previous publications [Bruening et al. 2003; Luk et al. 2005; Scott and Davidson 2001b].

Most dynamic translators operate by repeatedly translating instructions, then executing the translated instructions. An examplar, Strata, is shown in Figure 2. Each time the translator encounters a new instruction address (i.e., PC), it first checks to see if the address has been translated into the *code cache*. The code cache is a software instruction cache that stores portions of code that have been translated from the application text. The code cache is made up of *fragments*, which are the basic unit of translation. If the translator finds that a requested PC has not been previously translated, it allocates a fragment and begins translation. When an end-of-fragment condition is met (e.g., an IB is encountered), the translator emits any *trampolines* that are necessary. Trampolines are code segments inserted into the code cache to transfer control back to the translator. Most Control Transfer Instructions (CTIs) are initially translated to trampolines (unless its target is already in the code cache). Once a CTI's target instruction becomes available in the code cache, the trampoline is replaced with a CTI that branches directly to the destination in the code cache. This mechanism is called *fragment linking* and avoids significant overhead associated with returning to the translator after every fragment [Witchel and Rosenblum 1996; Cmelik and Keppel 1994; Bruening 2004; Scott and Davidson 2001b].

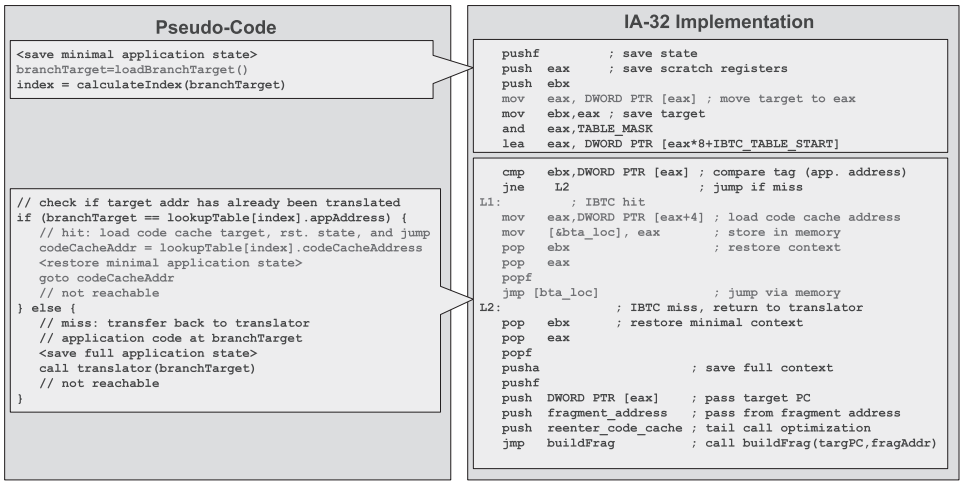


Fig. 3. IBTC lookup algorithm and corresponding implementation for the IA-32.

3. INDIRECT BRANCH TRANSLATION

A fundamental operation performed by an SDT system is translation of branch target addresses. The SDT system must map an application branch target address to the appropriate code cache address.

$$t(Addr_{Application}) \rightarrow Addr_{codeCache}$$

Translation of direct branches is simple because the mapping of target addresses is one-to-one and can be done at translation time. On the other hand, translation of IBs is more difficult. The mapping is many-to-many (each indirect branch maps an arbitrary number of application addresses to an equal number of code cache addresses) and must be done at execution time. The many-to-many nature of indirect branches necessitates that the SDT system generate efficient code to perform this mapping.

A variety of IB handling mechanisms have been used in SDT systems. The techniques can be classified into three main categories: data cache hashing (Section 3.1), instruction cache hashing (Section 3.2), and inline instruction cache handling (Section 3.3). For some types of indirect branches, namely return instructions, an SDT system can benefit from the predictable use patterns. Section 3.4 discusses some special mechanisms for translating return instructions, but Sections 3.1 through 3.3 first discuss the generic IB translation strategies.

3.1. Indirect Branch Translation Cache

An Indirect Branch Translation Cache (IBTC) is a data structure used to translate the target of an IB to the corresponding code cache address [Scott et al. 2003]. An IBTC is a list of pairs of addresses: an application address and its corresponding code cache address. The left side of Figure 3 shows pseudocode for performing an IB translation using an IBTC, and the right side of Figure 3 gives the Intel IA-32 implementation.

The IBTC lookup code first needs to save some architecture state, as the IB translation code cannot safely modify any registers. For the IA-32 instruction set, saving state includes saving the `eflags` via the `pushf` instruction. For the SPARC, saving state is more complicated because the current register window must be saved, and if the branch

target address is in the register window, it must be stored in a thread-safe temporary location before any `save` instruction.

After saving the context, the IB target is loaded into a temporary register. Another temporary is used to calculate the index into the IBTC table by masking based on the size of the table (`TABLE_MASK`). The appropriate IBTC entry is computed by adding the index to the base address of the IBTC table (`IBTC_TABLE_START`). The application target address is compared against the IBTC entry. If the entry matches the target address (an IBTC *hit*), the corresponding fragment address is loaded, the application state is restored, and control is transferred to the target fragment. If the entry does not match (an IBTC *miss*), control is transferred to the translator so the target fragment can be built and a new entry made in the IBTC.

There are many IBTC design options. One option is whether the system should reprobe on a conflict miss in the IBTC. The initial design of our SDT system's IBTC treated the IBTC much like a hardware cache, meaning that if there was a conflict miss, the translator would be invoked to replace the conflicted cache entry. If conflict misses are a large part of the remaining overhead, the IBTC can be implemented as a traditional hashtable and be reprobed on a conflict miss, thus reducing the cost of frequent conflicts.

Another important design choice is whether to use a single, large IBTC shared between all IBs, or to use a small fixed-size individual IBTC for each IB (nonshared). Empirical tests revealed that it was difficult to determine a good fixed size for individual IBTCs because some IBs only have a few targets, and therefore only require a small IBTC. Other IBs have many targets which could lead to a high number of conflict misses. This observation led to the idea of nonshared, adaptively sized IBTC. An adaptive IBTC doubles in size when a conflict miss occurs. This IBTC model allows IBs with few targets to remain small while avoiding conflict and capacity misses for IB translations with many targets.

3.2. Sieve

The sieve is a translation technique that uses code to map an application's IB target address to a code cache target [Sridhar et al. 2005]. The sieve is essentially an open hashing technique implemented solely with instructions [Sedgewick 1983]. The left side of Figure 4 gives the pseudocode for an implementation of the translation code that a dynamic translator would generate and place in the code cache. The right side of Figure 4 gives the SPARC implementation.

For each IB encountered during the translation process, the dynamic translator emits code to store the branch target address and transfer control to the sieve dispatch code (at label `SieveDispatchBlock`). The sieve dispatch code is created during initialization of the code cache. When executed, this code saves any application state required, reloads the application IB target address, and uses this address to calculate an index (using a simple mask) into the sieve jump table. The last instruction of the `SieveDispatchBlock` is an IB to an entry in the *sieve jump table* selected by the index.

The sieve jump table (at label `SIEVE_JUMP_TABLE`) contains jump instructions that jump either to a `ReturnToTranslator` block or to a *sieve bucket*. Each sieve bucket compares the application target address to a previously seen target address, and branches directly to the appropriate code cache destination if the targets match. If the match fails, control is transferred to the subsequent sieve bucket. The last sieve bucket in each chain contains a branch to the `ReturnToTranslator` block. This block is reached if the target of the IB target has not been translated. The `ReturnToTranslator` block invokes the dynamic translator to create a new sieve bucket entry and update the appropriate entry in the sieve jump table to jump to it. In the pseudocode presented, the third entry in the sieve jump table has been modified to jump to `SieveBucket01`.

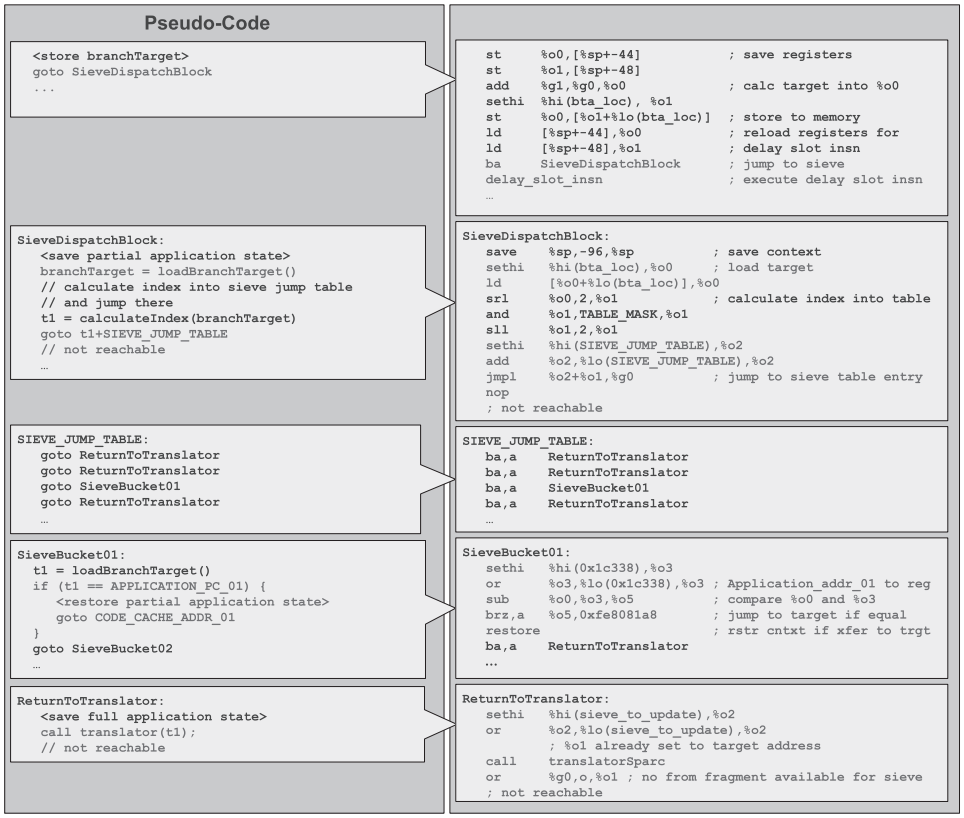


Fig. 4. Sieve algorithm and corresponding implementation for the SPARC.

Initially, all the jumps in the sieve jump table point to the `ReturnToTranslator` block. As IBs are processed, the translator fills the table with jumps to an initial sieve bucket. As with all open hashing implementations, the efficiency of the scheme depends on the keys (i.e., the IB target addresses) being uniformly distributed over the bucket table (i.e., the sieve jump table).

Like the IBTC implementation, the sieve implementation must be carefully crafted to be efficient. One interesting artifact to note is that the sieve never compares two address values. Instead, the sieve repeatedly compares one address value to a constant. Additionally, only one temporary is needed, so the overhead of the context save and restore may be lower than other techniques. Finally, it should be noted that the sieve uses code space while the IBTC uses data space and the efficiency of the techniques could be dependent on the relative amounts of I-cache and D-cache available on the target machine. Thus, some machines may perform better with the sieve than with an IBTC and vice versa.

3.3. Indirect Branch Inlining

Instead of relying exclusively on either an IBTC or sieve hash, it is possible to emit inline code to do a tag compare and appropriate transfer instruction sequence. Using one or more *inline cache entries* can be advantageous if many IBs resolve to a few targets most of the time. IB inlining is done by comparing the IB target to the inlined target address. If the target does not match, the inlining code can be followed with

```
<save minimal application state>
t1 = loadBranchTarget()
if(t1 == STORED_TARGET) {
   <restore minimal application state>
   goto CODE_CACHE_ADDRESS
}
```

Fig. 5. Pseudocode for IB inlining.

another such inline, or another IB handling mechanism. If the hit rate of an inline cache entry is high enough, execution of the inline cache entry (which is shorter than a full data or instruction hash lookup) can save significant time and cache pollution. The pseudocode for IB inlining is shown in Figure 5. STORED_TARGET is the application address corresponding to the CODE_CACHE_ADDRESS in the code cache. Since this target address is potentially unknown at fragment-creation time, the SDT system creates an empty code template until the decision can be made about how to use inline cache entries.

There are a number of parameters to consider when using inline cache entries. First, how many inline targets should be used? Is a fixed number of inline cache entries appropriate, or should the amount of inlining be dynamically determined? How many times should the IB translation be executed before the inlined targets are selected? Is it advantageous to handle IBs resulting from indirect calls differently from IBs resulting from indirect branches? Section 5.4 empirically evaluates these options.

3.4. Translating Return Instructions

Return instructions are the most frequently executed type of indirect branch. It has been long known that most programs use function return statements in predictable ways. Nearly every return statement transfers control to the location immediately after a corresponding call statement. Many architectures, such as IA-32, SPARC, and MIPS, exploit this pattern via special instructions to perform function call and return operations. Hardware designers implement highly accurate branch predictors that achieve near-perfect prediction rates for return instructions using a return address stack [Skadron et al. 1998]. SDT systems can also take advantage of this pattern predictability to reduce the overhead of maintaining control when mimicking the effects of a return instruction. Sections 3.4.1 through 3.4.3 examine three mechanisms that exploit the predictable patterns of return instructions.

3.4.1. Return Address Translation Stack. The *Return Address Translation Stack* (RATS) is a mechanism that exploits the predictable nature of return instructions much like a hardware return address stack [Hazelwood and Klauser 2006]. The translation of a call instruction pushes an application and fragment cache return address into a buffer. A return's translation pops these values and verifies their validity and transfers control appropriately if they are correct. If incorrect, a generic IB translation technique is used as a backup mechanism.

Figure 6 shows the pseudocode for the RATS. Note that each call/return pair requires writing and reading two pointers to the RATS. Further, since the table is maintained as a stack, an update of the stack's head pointer needs to be performed for each call and return. Checking for over- or underflow can be achieved by allocating a page for the RATS and using hardware page protections to disallow read or write accesses to adjacent pages.

In general, because of the high overhead associated with maintaining the stack pointer and instrumenting both the call and return instructions, this mechanism has

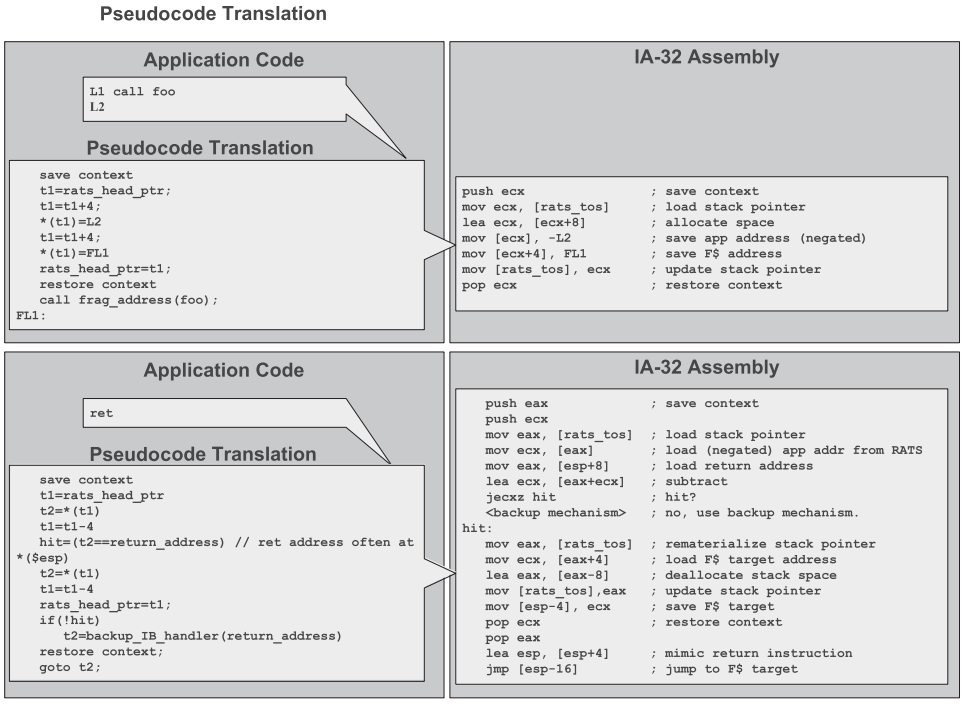


Fig. 6. Pseudocode and IA-32 assembly for the RATS.

significant overhead even though it can have a very high hit-rate. An appropriately sized sieve or IBTC is generally faster. The next section describes several optimizations on the RATS to make it more competitive with generic IB translation techniques.

3.4.2. Return Cache. The *return cache* is a mechanism to exploit the regularity of return instructions, first presented by Sridhar [2005]. In the return cache implementation, a table of fragment cache return addresses is maintained (accessed by a hash function using the containing function's entry point). Figure 7 shows the pseudocode for the translations of an application's call and return instructions, along with the IA-32 and SPARC assembly code. As the figure shows, the translation for each call instruction updates the table and return instructions are translated to use the value in the table. A return instruction's translation does not verify that the value in the table is correct when the translation executes. Instead, the translation blindly jumps to the location specified in the table. In most cases, this will be the correct address. In some cases (such as recursive function calls, odd uses of return instructions, or cache collisions) the fragment return address in the table will be incorrect. To maintain proper program semantics, the instructions at the destination verify that the control transfer was correct (empty entries in the table jump to a block which uses a generic backup mechanism). Since the address is frequently correct, this verification can be done very inexpensively using an instruction sequence similar to an inline cache entry. If the address is not correct, the branch is treated as a generic indirect branch via a backup mechanism such as the sieve or IBTC.

This mechanism, in essence, uses a software cache to cache the top of the RATS at the expense of a possibly higher miss rate. However, using a cache instead of a stack avoids having to update stack pointers. Instead, indexing into the cache can be performed at translation time and avoids repeatedly performing these operations at

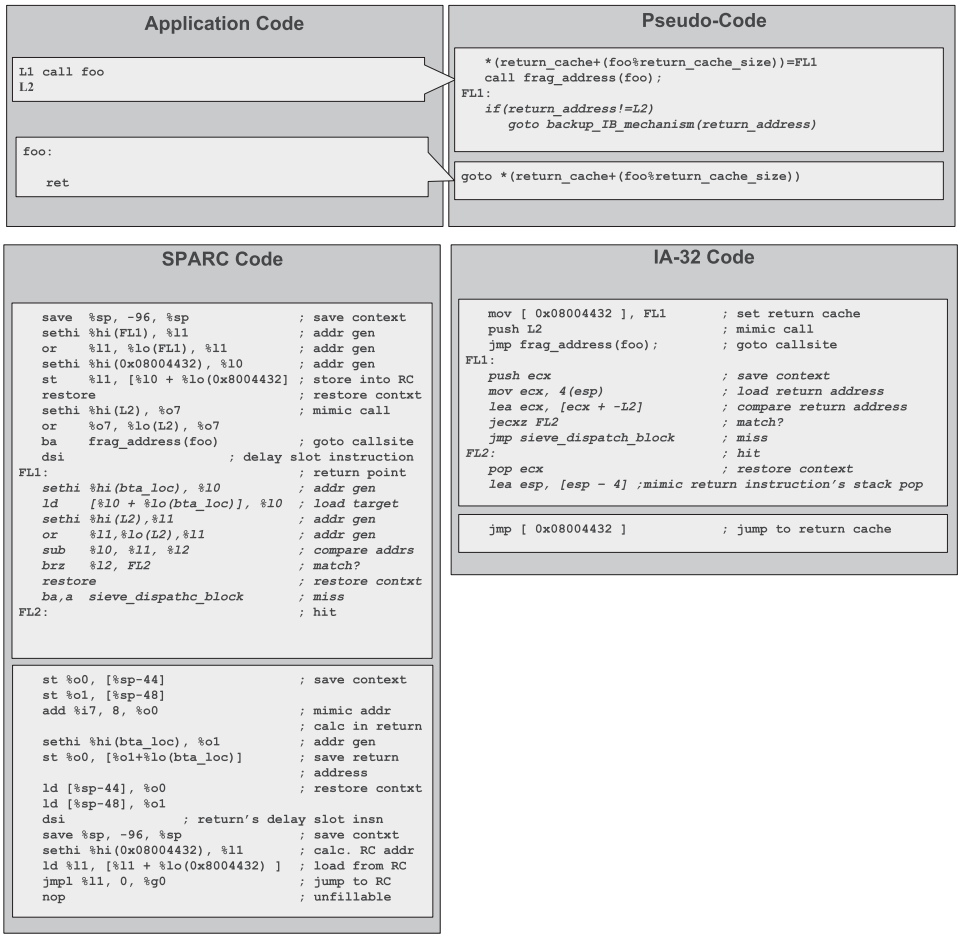


Fig. 7. Pseudocode and assembly for the return cache. Italicized parts represent the inline cache entry to verify the return address is correct.

fragment-execution time. Likewise, by speculatively jumping to the fragment return address, the need to store the application return address at the call site can be avoided.

3.4.3. Fast Returns. An alternate mechanism to using a RATS or return cache is to use a mechanism called *fast returns*. Instead of exploiting the pattern of calls and returns, fast returns exploits the fact that a return address generated by a call is rarely used except by a return instruction.

Instead of translating a call instruction to push the application's return address, the fragment address is stored into the return address location. Return instructions are left untranslated to simply return to the fragment cache return address and continue execution unchecked. Such a translation avoids much, if not all, of the overhead associated with translating return instructions.

For the majority of call/return pairs, the fast return technique is effective because the return address is only used by the return instruction. There are a few cases where the address stored by a call instruction is used differently, and must be handled as a special case if fast returns are used. One example of this behavior is data access for position-independent code in some versions of glibc for IA-32 machines. Data is accessed based

on an offset from the current PC, which is not known when the code is generated. To identify the current PC, the system calls a *thunk*, which simply loads the return address into a given register and then returns, thus placing the callee's PC into a predetermined register. If fast returns are implemented, the register loads a fragment cache address and the address of the associated data is calculated incorrectly. Fortunately the code for a thunk function is easy to identify when the fragment is translated. The SDT is able to identify the thunks and place the application address in the register directly.

Two other constructs that must be handled for the safe execution of fast returns are C++ exceptions and `setjmp()`/`longjmp()` functions. These constructs alter control flow from one function to another function somewhere up the call chain. One way to implement such a feature is to use the return address values to "walk the stack" up to the desired stack frame. If the return address is a fragment address, the walk up the stack can be corrupted and cause the program to fail. Like the thunk code, this behavior is recognizable at translation time, and the SDT can emit code to fix the stack when an exception or long jump occurs.

4. EXPERIMENTAL PARAMETERS

To evaluate the mechanisms for handling IBs described in Section 3, we used a variety of machines, compilers, and benchmarks. The techniques described in Section 3 were implemented and evaluated within the Strata dynamic binary translation framework [Scott and Davidson 2001b]. Sections 4.1 and 4.2 discuss our assumptions, machines, compilers, compiler options, and benchmarks we used for evaluation purposes.

4.1. Machines

The techniques presented in Section 3 were evaluated on three platforms: an Intel Pentium 4 Xeon [Intel 2005], a Sun UltraSPARC-IIi [Sun Microsystems 1997], and an AMD Opteron 244 [Advanced Micro Devices 2006]. The Pentium 4 Xeon data cache is 8KB and 4-way associative, while the instruction caching mechanism is an 80K micro-op trace cache. The UltraSparc-IIi data cache is 16KB direct-mapped, while the instruction cache is 16KB, two-way set associative. On the Opteron both the data and instruction caches are 64KB, two-way associative.

The compiler used on the Pentium 4 Xeon is `gcc` version 3.3. On the Opteron the compiler is `gcc` version 4.0.2. For both of these machines, the compiler options are `-fomit-frame-pointer -O3`. The compiler used on the UltraSPARC-IIi machines is the SUNWspro `cc` compiler using options `-fast -xO5`. Strata was configured with a 4MB code cache which was sufficiently large to run all applications.

4.2. Benchmarks

The full set of SPEC CPU2000 benchmarks was used for evaluation in this work [Standard Performance Evaluation Corporation 2011]. Since all benchmarks have a significant number of return instructions per second, we use the full suite when evaluating return handling mechanisms. For mechanisms applied to nonreturn instructions, most graphs report results (the SPEC number as reported by `runspec` for three runs) only for the SPEC benchmarks that execute a significant number of IBs, namely: `177.mesa`, `176.gcc`, `186.crafty`, `252.eon`, `253.perlbmk`, `254.gap`, and `255.vortex`. For these graphs, we also report the geometric mean (ave) of the seven benchmarks. For graphs that do include the entire SPEC suite, we report the geometric mean of the SPEC number for the integer benchmarks (int ave), floating-point benchmarks (fp ave), and the entire SPEC CPU2000 suite (spec ave). All results are normalized to native execution time (i.e., the execution time of the application not running under SDT control).

Table I. IB Handling Mechanisms and Configuration Choices

Mechanism	Choice	Description
IBTC	Table size	Maximum number of target address translations in the table
	Lookup placement	Lookup code placed in each fragment or in a separate function
	Reprobing	Check next entry for a translation on a conflict miss
	Adaptively sizing	Grow IBTC on a conflict miss
	Sharing	One table for all indirects or separate tables for each indirect
Sieve	Size	Number of buckets in the sieve
Inline	Inline amount	Maximum number of target address translations to inline in the fragment
	Target selection	Sequence of inlined entries ordered by time or frequency (earlier/hotter translations occur earlier in the sequence)
	Profiling	Pre-computed from a previous run with the same data set (ideal) or online during a short sampling period (online profile)
	Indirect type	Determine inline amount, target selection, and profiling amount separately for indirect calls and other indirect branches
	Fall back	Mechanism to handle a miss on all inlined translations
Return Cache	Size	Number of entries in the return cache table
Fast Returns		No options
RATS		No options

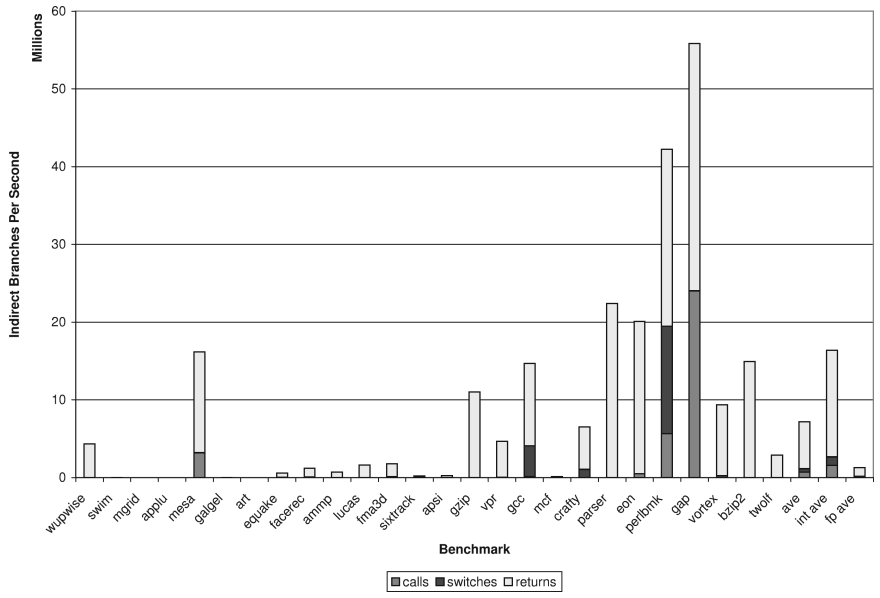


Fig. 8. Indirect branches executed per second on Pentium 4 Xeon. Note that the bar for call instructions represents only indirect call instructions.

5. EXPERIMENTAL RESULTS

This section evaluates the different IB handling techniques discussed in Section 3. Table I summarizes the configuration options for each IB handling mechanism.

Figure 8 shows the indirect branch rate for return, and call instructions, and the remaining indirect branches (mostly for switch/case type statements). The figure shows

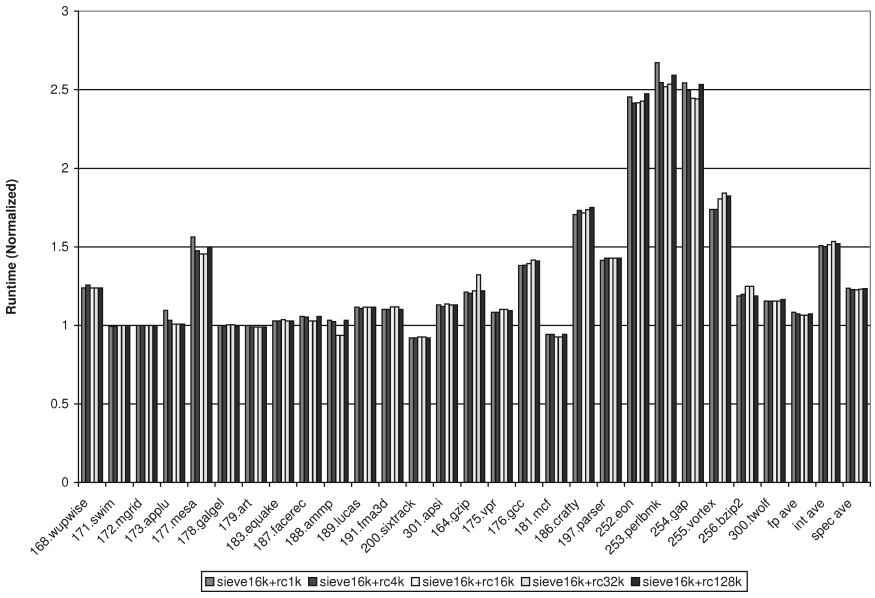


Fig. 9. Return cache size on UltraSPARC-IIi.

that return instructions are the most common type of indirect branch in most applications. Consequently, we choose to study them first.

5.1. Return Handling Mechanisms

Section 5.1.2 compares the return handling mechanisms (RATS, return cache, and fast returns), but first it is necessary to find the proper size for the return cache in Section 5.1.1.

5.1.1. Return Cache. We first evaluate the size of the return cache. Figure 9 shows the performance of Strata on an UltraSPARC-IIi machine, configured with a 16K-entry sieve to handle nonreturn instructions (which we later show to be satisfactory for handling nonreturn instructions). The size of the return cache varies from 1K entries to 128K entries. As the figure shows, there is little performance gain as the size of the table reaches 4K entries. Some benchmarks start to show performance degradation as the size grows, relating to data cache pollution. When the table size reaches 128K entries, the constant needed to index the table exceeds the immediate field of the SPARC's instruction set. Consequently, updating the table for indirect call instructions takes an extra instruction, and some slowdown is observed for benchmarks with many indirect calls (`253.perlbmk`, `254.gap`, and `177.mesa`).

We considered not updating the return cache for indirect call instructions because the update is significantly more expensive than updates for direct call instructions. However, skipping these updates would likely cause the corresponding return instruction to miss in the return cache, and fall back to the backup indirect branch handling mechanism (the sieve in this case). Experimentally, we found that not updating caused minor slowdown in benchmarks with indirect call instructions (detailed results omitted).

Overall results for the Pentium 4 Xeon are shown in Figure 10. Results generally trend the same as for the UltraSPARC-IIi, with the exception of the 16K-entry return cache. For the 16K-entry return cache, saving and restoring the `eflags` register can be avoided when calculating the index into return cache for indirect call instructions.

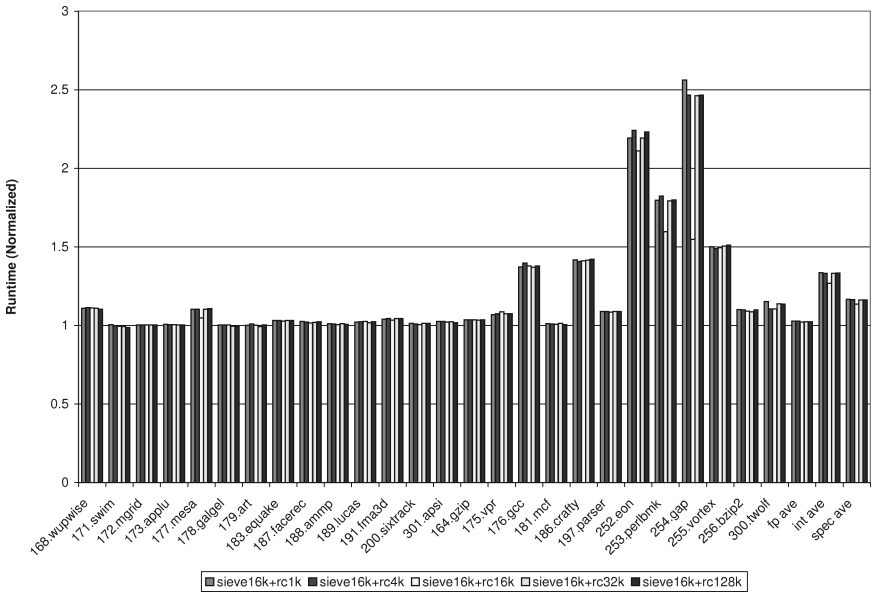


Fig. 10. Return cache size on Pentium 4 Xeon.

Avoiding saving/restoring of flags is discussed more in Section 5.3. Results on the AMD Operton are the similar and are omitted.

5.1.2. Comparison. Figures 11 and 12 show the comparison between the return cache, fast returns, the sieve, and the IBTC for the UltraSPARC-IIi and Pentium 4 Xeon, respectively.

On the UltraSPARC-IIi, the tuned return cache is more expensive than simply using a sieve or IBTC. The main reason for the poor return cache performance is the SPARC's ISA's limited addressing modes. Because there is no support for pointer-sized (32-bit) immediates, it becomes necessary to save a register and use multiple instructions to generate the address to store into the return cache (see Figure 7), and likewise, to load the value to which to return takes a spare register and multiple instructions. Lastly, the pointer-sized comparisons to verify the return cache entry take a spare register and extra instructions. Consequently, there is significant extra overhead associated with using the return cache.

However, the IA-32 machines support loading and storing a pointer-sized constant using an immediate address in just one instruction. Because of the machine's efficient support for the operations to update and access the return cache's table, performance for these machines is greatly enhanced. The benchmarks with the most return instructions per second (`177.mesa`, `253.perlbmk`, `254.gap`, and `252.eon`) see the largest improvements.

On all machines, though, fast returns vastly outperform the return cache (Operton 244 results are omitted). Such performance is not unexpected: fast returns yields no overhead for most call/return pairs. Experiments using hardware performance counters indicate that instruction count is greatly reduced. Further, we find that branch predictor performance is improved; the hardware is able to use its hardware return address prediction mechanism for return instructions. For the SPEC benchmark suite, the potential loss-of-control or incorrect execution issues with fast returns never was a problem, demonstrating that for some benchmarks, fast returns are a viable option.

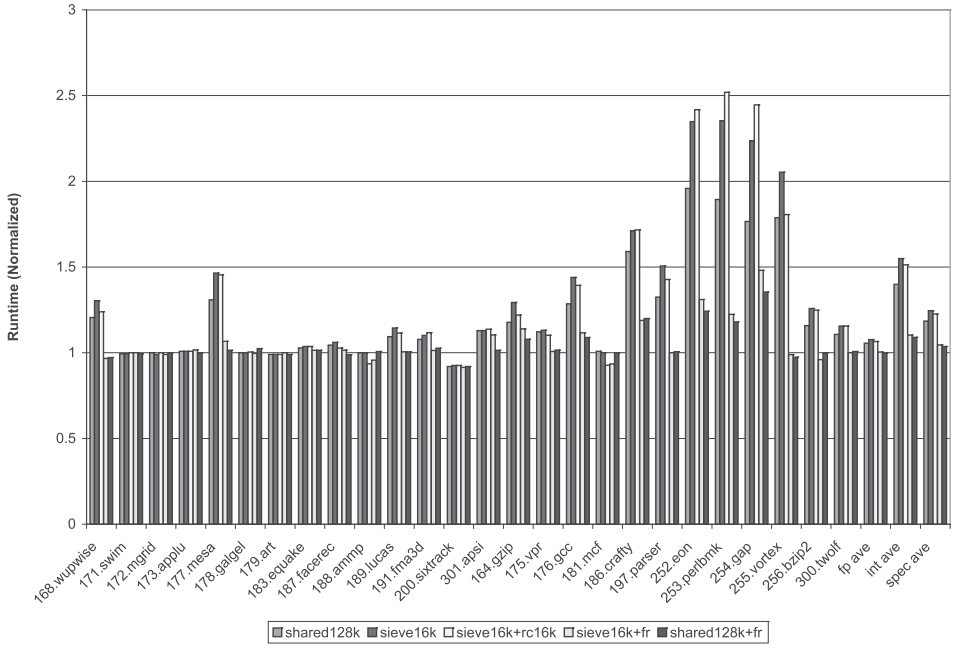


Fig. 11. Comparison of return handling mechanisms on UltraSPARC-IIi.

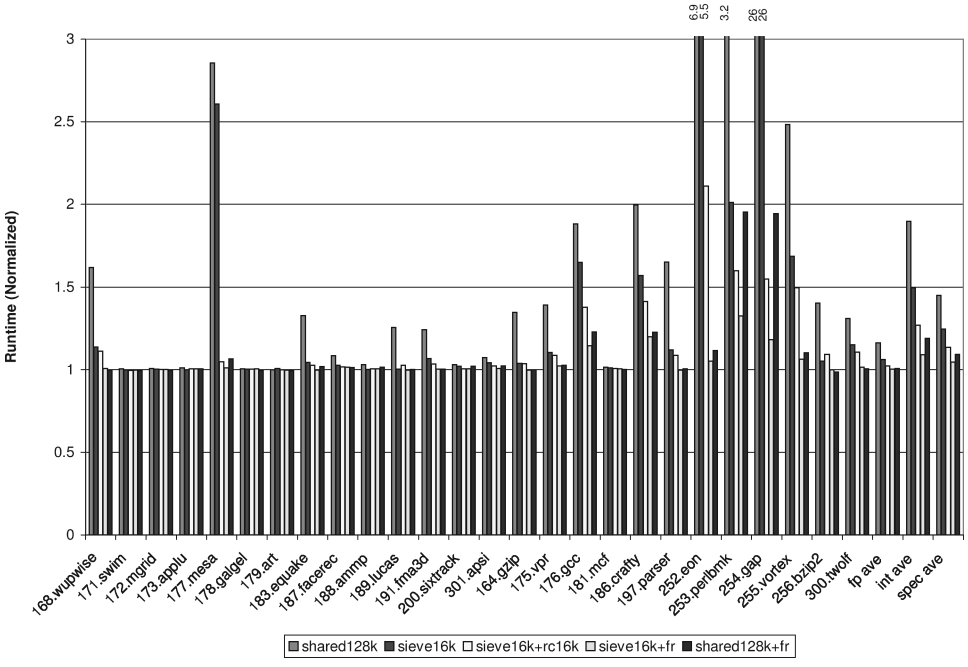


Fig. 12. Comparison of return handling mechanisms on Pentium 4 Xeon.

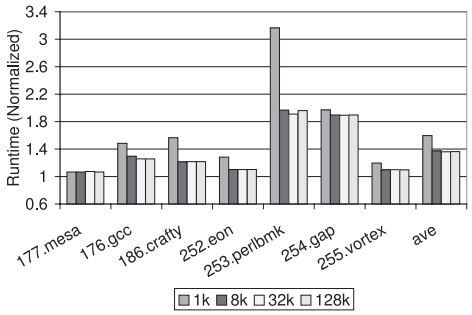


Fig. 13. Performance with varying IBTC sizes (Pentium 4 Xeon).

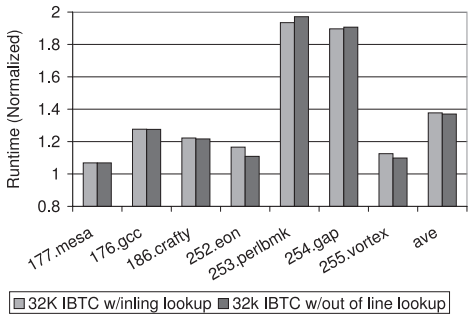


Fig. 14. Placement of code to perform IBTC lookups (Pentium 4 Xeon).

As the return cache is an optimzation on the RATS, we expect the return cache to be more effective. To verify this conclusion, we implemented the RATS for the IA-32 instruction set, and found it was 7% slower than the return cache. Though hit rates for the RATS mechanism were near 100%, often with only one miss even in the reference runs of the program, the high instruction count made the mechanism less efficient.

The remainder of the experiments use the fast return mechanism for return instructions.

5.2. IBTC

5.2.1. Table Size. First, an appropriate size for a shared IBTC was determined by increasing the size of the IBTC table until performance ceased to improve. Figure 13 shows the results for the Pentium 4 Xeon. As the figure shows, a small IBTC table yields very poor performance, especially in `253.perlbmk`. With 8K entries, nearly all of the performance gain is realized. At 32K entries, no additional performance improvement is observed. Results are similar on the UltraSPARC-IIi and Opteron 244 and are omitted due to space constraints.

5.2.2. Lookup Code Placement. Second, we determined if the placement of the IBTC lookup code affects performance. Figure 14 shows the results of placing the lookup code inline in the fragment (first bar) versus creating an out-of-line function to perform the lookup (second bar) on the Pentium 4 Xeon. The data shows that some benchmarks benefit slightly from placing the lookup code in a separate function. Other benchmarks show slight degradation in performance. In general, lookup code placement is a time/space trade-off. The best choice depends on the constraints of the system and the properties of the individual benchmark. On average, no significant difference between inline code and using an out-of-line function can be seen. This result holds on the

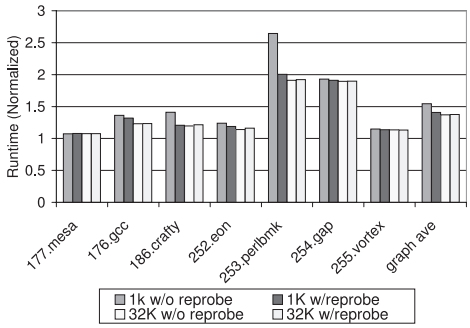


Fig. 15. Performance of IBTC reprobing on conflict (Pentium 4 Xeon).

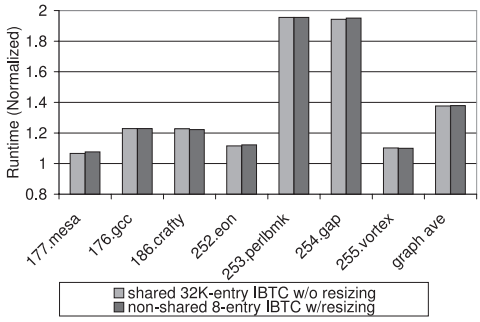


Fig. 16. Performance with adaptive resizing of IBTC (Pentium 4 Xeon).

AMD and Sun machines and the detailed results are omitted. Subsequent experiments evaluate IBTC options using inline IBTC lookup code.

5.2.3. Reprobing. Next, we considered the benefit of reprobing the IBTC to handle conflicts. Figure 15 shows the performance on the Pentium 4 Xeon of reprobing a 1K-entry IBTC (first two bars) and 32K-entry IBTC (second two bars). The data shows that reprobing provides significant benefits to a 1K-entry IBTC, nearly equaling the performance of a 32K-entry IBTC. For large IBTC sizes, however, reprobing provides little benefit. Consequently, we believe that reprobing would be a beneficial addition to any IB handling mechanism in a space-constrained system [Moore et al. 2009]. Results on the UltraSPARC-IIi and Opteron 244 are similar and are omitted. Since we use an IBTC large enough to avoid most conflicts, we do not enable the reprobing mechanism for the subsequent IBTC experiments.

5.2.4. Resizing and Sharing. Reprobing the IBTC seems to be an efficient way to deal with conflicts as long as conflicts are rare. It is possible to completely avoid all conflicts by resizing an IBTC when a conflict is detected. Figure 16 shows the results of assigning a small IBTC (8 entries) to each IB and doubling the size of the IBTC on any conflict for the Pentium 4 architecture. The figure shows that adaptively resizing an IBTC adds little benefit over a large nonadaptive, shared IBTC. Another experiment, not shown, set the initial size of an individual, adaptive IBTC much larger (512 entries). No performance benefits were noted.

To summarize, we find that a large (32K entries), shared IBTC with inline lookup code provides the greatest reduction in overhead. Having nonshared, or adaptively sized IBTCs yields little benefit. Using an efficient reprobe mechanism can effectively deal with conflicts in a smaller (1K entries) IBTC.

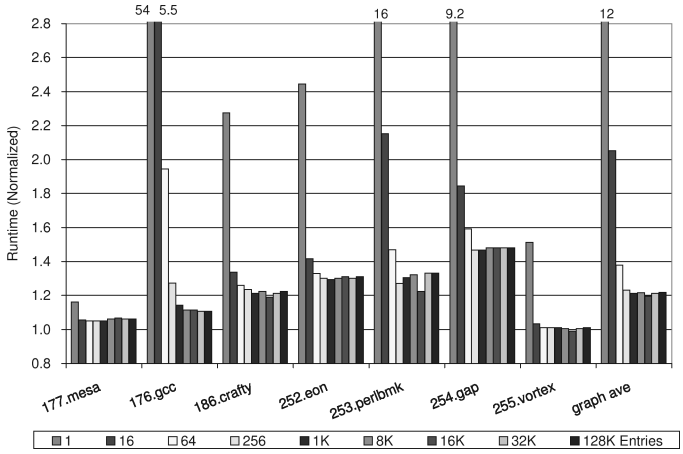


Fig. 17. Sieve size experiment (UltraSPARC-IIi).

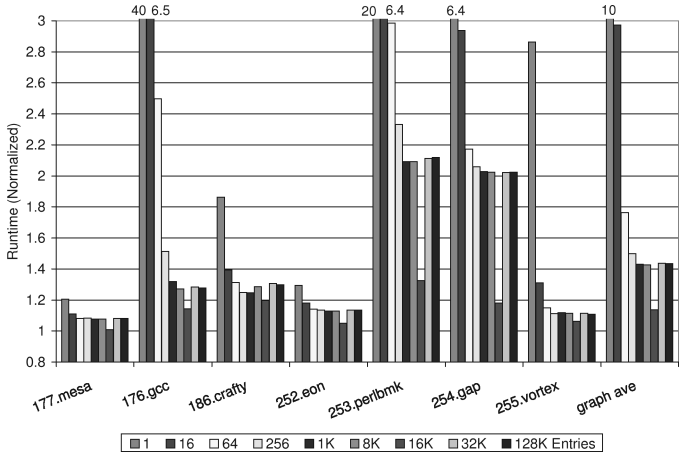
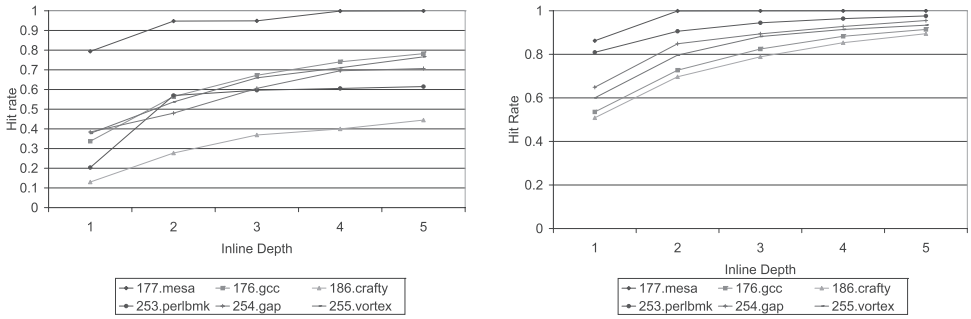


Fig. 18. Sieve size experiment (Pentium 4 Xeon).

5.3. Sieve

The sieve, as described in Section 3.2 and in previous work [Sridhar et al. 2005], has been implemented for all the machines we use for evaluation. We evaluated sieve sizes from 1 entry up to 128K entries. Figure 17 shows the results obtained on the UltraSPARC-IIi. The data shows that there is no further performance gain after 1K entries. In some cases, performance degrades after 1K entries because the `and` operations (see Figure 4) are more expensive. Due to the size of the `TABLE_MASK` and restrictions on the size of an immediate in an instruction, extra instructions are required to generate the constant for the operation.

Figure 18 shows the results on the Pentium 4 Xeon. Like the UltraSPARC-IIi, performance levels off after 1K entries, with one notable exception. The 16K-entry sieve achieves significant performance improvement. `254.gap`'s overhead improves from 2.02 times longer than native execution for the 8K-, 32K-, or 128K-entry sieve to only 18% longer than native execution for the 16K-entry sieve. The 16K-entry sieve performs significantly better because the save and restore of the `eflags` register can be avoided. Being able to avoid the save/restore of the flags is a result of being able to use the



(a) inline first target (b) inline best target

Fig. 19. Inline hit rate for 1 to 5 entries (Opteron 244).

`movzwl` instruction (which extracts the lower 16 bits of a register for zero-extending a 16-bit quantity, yet does not affect the flags) for the masking operation instead of the `and` instruction (which does modify the `eflags` register). On the Pentium, the cost of the save and restore of the flags is considerable, the 16K-entry sieve provides significant benefits. Results on the AMD Athlon show the same trend as the Xeon and are omitted due to space constraints.

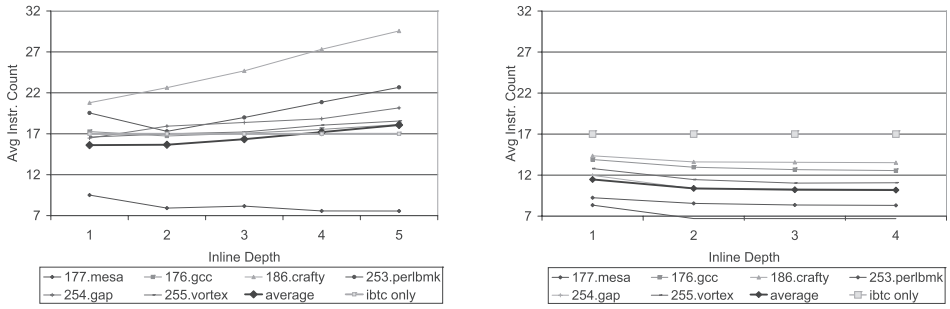
Since the sieve implementation already supports efficient reprobing, choosing the correct size is the only configuration parameter to consider. To summarize, a 1K-entry sieve for the UltraSPARC-IIi is best as it avoids the extra instruction for the `and` operations, while a 16K-entry sieve is best on the Opteron and Xeon as it avoids the use `pushf` and `popf` instructions.

5.4. Inline Cache Entries

The next option is how to use inline cache entries. Figure 19 shows hit rates of inline cache entries on the Opteron using two policies for selecting the inline entry. The left graph in the figure shows the hit rates when the first dynamically executed target(s) is used in the inline cache entry(s). The right part shows the hit rates when the best choice (most frequently executed) for an inline target is inlined. Our results (not shown) indicate that naïvely choosing the first target causes over 60% of all cache accesses in the integer benchmarks to miss when one address translation is done as an inline cache entry. By using the ideal inline targets (which obviously are not available at translation time), the inline entries miss rate drops to just 40% for one inline target. The floating-point benchmarks do much better, but these benchmarks have few IBs, so even a very high hit rate does not affect overall performance.

Since one inline entry does not have a high hit rate for some benchmarks, it is possible that inlining more entries may increase the overall hit rate. To get a sense of how the number of inline entries affects hit rate, hit rates for various amounts of inlining, ranging from 1 to 5 entries, were measured. Figure 19 shows the hit rate generally increases as more entries are inlined using the naïve target selection method. For example, in `177.mesa`, the hit rate quickly approaches 100%. In other benchmarks, with a greater distribution of indirect targets, the hit rate does not increase as dramatically. In `253.perlbmk`, the hit rate largely levels off after the second inlined entry. Even when using the ideal target selection mechanism and five inline targets, hit rates do not always reach 90%.

Although hit rates indicate how well inlining may work, the location and instruction cost of which inline cache entry the hit occurs also affects performance. In the worst case, the last entry in the inline sequence might be hit. Hitting in the last inline cache



(a) inline first target (b) inline best target

Fig. 20. Average instruction count for 1 to 5 inline entries (Opteron 244).

entry is expensive because several entries have to be traversed before getting to the last one.

Figure 20 shows the average number of instructions needed to satisfy an IB translation for inline depths of 1 to 5 on the Opteron using both target selection mechanisms, based on the Instruction Count (IC) of hitting a particular inline entry and its local hit rate. Both sieve and IBTC lookups take approximately 17 instructions, depicted as a flat line in the figure. As shown, inlining reduces IC as compared to the IBTC on several benchmarks. For instance, `177.mesa` has an improvement. In other cases, the change in IC is small. In `253.perlbmk`, the count increases. On average, inlining one entry takes 14.6 instructions and inlining two entries takes 13.3. The relative change is small, and beyond two entries, there is little apparent benefit.

Because ideal target selection is impossible to implement a priori, we need to determine if an approximation to this scheme is worthwhile. To measure the effect of ideal target selection approximation, dynamic instruction scheduling, caching effects, and other machine considerations, we implemented an online profiling technique for IBs. The branch translation is executed a specified number of times, and then the best (most frequently executed during the online profiling) target(s) are selected and inlined. Since performing online profiling takes time, we considered using online profiling up to 300 executions of each IB. Beyond that point we began to see performance degradation and concluded no further online profiling was beneficial. Based on the results shown in Figure 19 and 20, we consider inlining 0 to 3 targets. Furthermore, we also considered call-type IBs separately from switch-type IBs as they may demonstrate different profiling and inlining behavior. Lastly, we considered allowing the translator to dynamically choose (based on the instruction count of inline cache entries and the backup mechanism) the number of inline entires once the profile selection is complete. Thus, if the profile indicates there are no dominate IB targets, the translator is free to rewrite the IB translation to use no inline entries, effectively avoiding any further overhead from frequently missing in the inline cache entry.

The detailed results (omitted due to space restrictions) indicate that call-type and switch-type IBs indeed show different behavior. The most beneficial option for switch-type instructions is to profile a modest amount, about 30 executions of the IB, and then allow the translator to choose the amount of inlining. Indirect calls seem to have more static behavior and inlining the first two targets provides the best speedup. For indirect calls, profiling, even a small amount, provides no additional benefits. Dynamically selecting the amount of inlining provided no benefit for call-type IB translations.

Figure 21 compares no inlining, the best switch-type inlining, the best call-type inlining, and the combination of the best switch and call inlining when using the

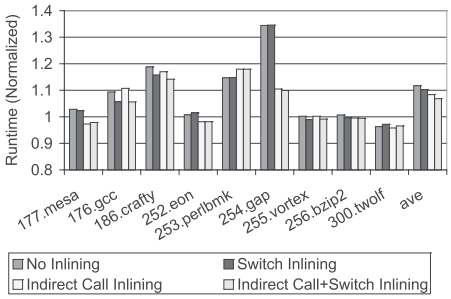


Fig. 21. Inlining results for Opteron 244.

sieve as a backup translation mechanism on the Opteron. The figure shows that switch inlining can provide modest improvements, about 2% on average for the key benchmarks while some benchmarks see up to 4% improvement. Call inlining provides more benefit, 5% on average. Using a call inlining technique for `254.gap` provides quite dramatic improvement on the Opteron. Experiments using hardware performance counters (omitted) demonstrate that this improvement is because the Opteron has a high misprediction rate for IBs, and the call inlining helps the processor mispredict the target less frequently.

Even though the indirect call and switch inlining techniques are apparently independent, the combination of the two techniques frequently yields performance worse than call inlining alone. Hardware performance counter experiments (omitted) show that this performance degradation is because inline entries take up significant instruction cache space. Combined, their potential gain in performance is erased by the increased instruction cache miss rate.

Interestingly, the Opteron is the only machine where using inline cache entries yields significant benefits. We find that the same configuration of inlining is best on the UltraSPARC-IIi and Xeon processors, but the use of inlining seems to yield little benefit. In fact, since the SPARC ISA has no instructions which support 32-bit constants, the inline entry requires extra instructions to generate the 32-bit addresses needed for the inline entry. Consequently, the inline cache entries are too expensive, and actually cause significant slowdowns for some benchmarks. The Xeon, which does support the 32-bit constants like the Opteron processor, sees no significant benefits; average results vary by less than 1%. We believe the Xeon does not see the benefits of the Opteron because the Xeon has a more sophisticated branch/trace predictor and the Pentium's trace cache is less tolerant to the increase in instruction cache pressure caused by the inline cache entries.

One last option that we considered was having the inline miss code (the sieve lookup code in the case of the Opteron) overwrite an inline entry. We found that this indeed provided a higher hit rate for the cache inline entry, but significant slowdowns overall, on the order of 2 to 5 times slower than native execution. The problem with dynamically updating the inline entries frequently is that the instruction cache must be flushed for every update of an inline entry. A single, frequently executed branch with a target that changes often will cause cache flushes too frequently for the instruction cache to ever perform well. Consequently, we disregard dynamic updating of inline cache entries once the initial targets have been selected.

To summarize, we find that cache inline entries can sometimes be effective at reducing overhead in dynamic translation systems. A small amount of online profiling, 30 executions, is needed for switch inlining to dynamically select the best amount

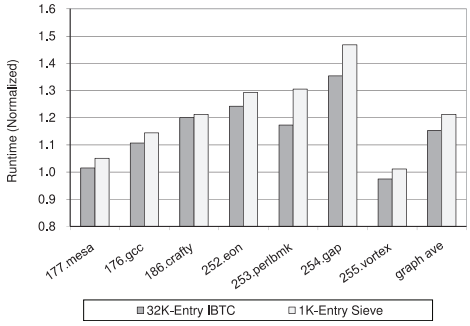


Fig. 22. Sieve vs. IBTC on UltraSPARC-IIi.

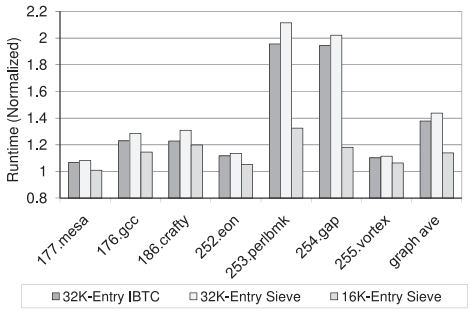


Fig. 23. Sieve vs. IBTC on Pentium 4 Xeon.

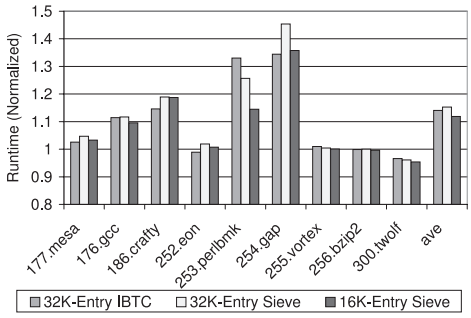


Fig. 24. Sieve vs. IBTC on Opteron 244.

of inlining. Call inlining should be performed using the first two dynamically taken targets, avoiding the profiling overhead.

5.5. Comparison

Now that we have explored the design and parameter space for IBTCs, the sieve, and branch target inlining, we examine which mechanism is most appropriate for each machine. For this evaluation, we compare the execution time of the benchmarks with different IB translation mechanisms, as well as the memory overhead for each technique. Other factors may be of importance in some systems. Due to space restrictions, we only consider performance and memory overhead.

Figures 22, 23, and 24 compare an IBTC to a sieve implementation on the UltraSPARC-IIi, Opteron 244, and Pentium 4 Xeon, respectively. For the UltraSPARC-IIi, the sieve has worse performance than the IBTC. The SPARC's limited addressing

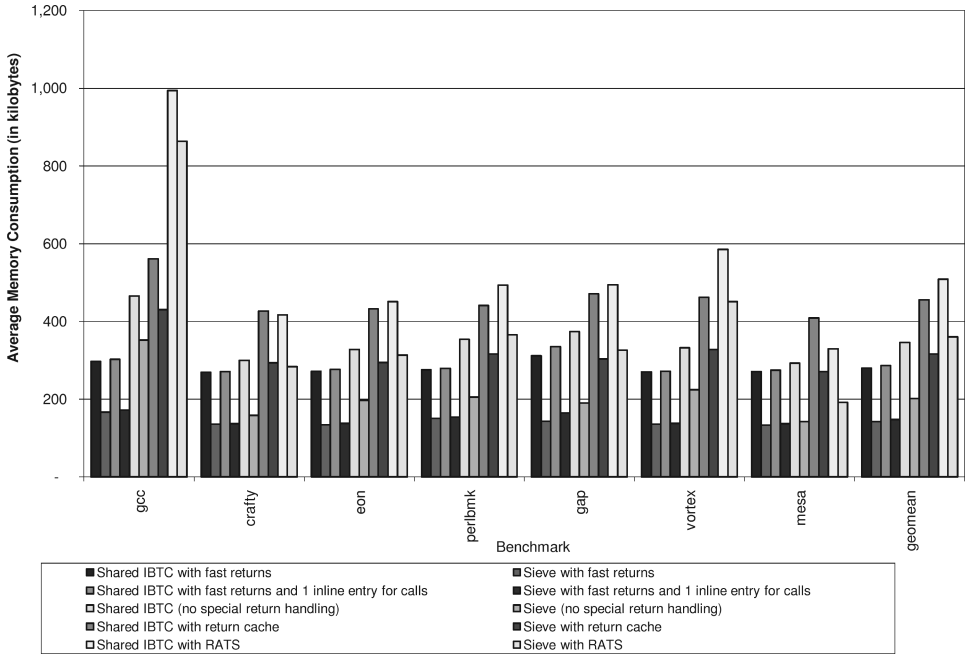


Fig. 25. Memory sizes for different IB techniques for the Opteron 244.

modes and fast context saves/restores make the sieve ineffective. The Xeon machine shows that the sieve would be a loss, except that the ability to avoid saving the flags provides a large win. Interestingly, on the Opteron 244, the sieve and IBTC perform similiarly on most benchmarks when the flags are saved. However, like the Xeon, once the save/restore of the flags is avoided the sieve performs much better.

Figure 25 shows the estimated memory consumption of each technique. The estimate includes exact information for all data required for the technique. The size also includes an estimate of the size of code (held in the fragment cache) for IB translations. Code size is estimated based on event counts (such as sieve transfer blocks generated) multiplied by the typical size of the translation. Code size estimates may slightly differ from the exact code size because the size of assembled instructions may vary due to different addressing modes, etc.

The first two bars of the figure show a shared IBTC (32K-entry) and a sieve (16K-entry). For both bars, the total size is dominated by the hash table required for the technique. Consequently the sieve has a smaller memory footprint due to its smaller table size. The next two bars show the overhead of using one inline cache entry for each indirect call translation, as detected to be most beneficial in Section 5.4. The memory size of the inline cache entries is insignificant compared to the size of the backup mechanism used. On average, the inline cache entries consume only 3.6K of code space.

The fifth and sixth bars in the graph represent the total size needed when the shared IBTC and sieve, respectively, are used without the fast return mechanism. We see that there is significant growth in the size required, even though the table sizes remain the same. This growth corresponds to longer IB translations for return instructions, as well as more sieve transfer blocks. Since there are a significant number of return instructions translated, these longer translations and more sieve transfer blocks result in this significant overall increase in memory consumption. It is interesting to note that

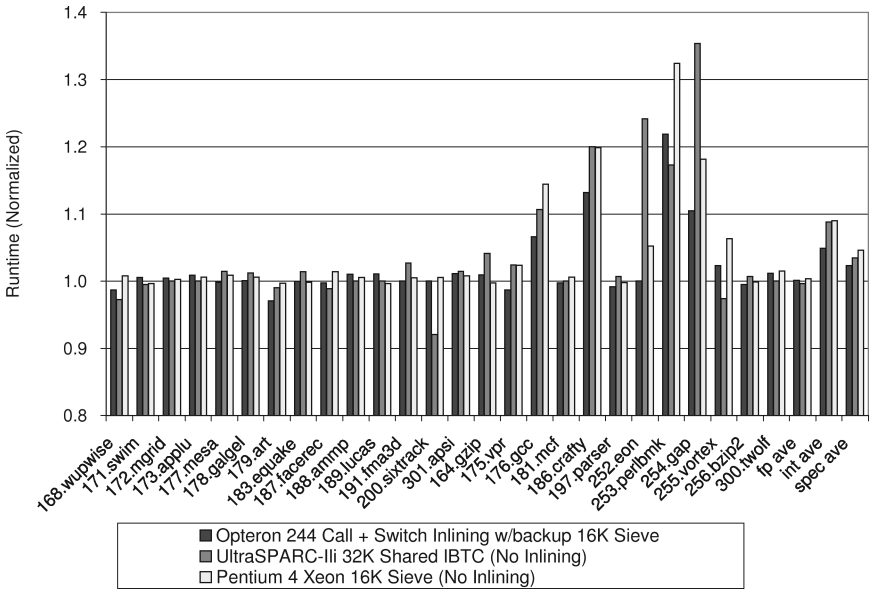


Fig. 26. Performance with best configuration.

the size difference between the shared IBTC and the sieve remains approximately the same as when using fast returns. This similar growth can be explained by the IBTC having a larger IB translation, while the sieve requires additional transfer blocks. These two differences ultimately require approximately the same amount of memory.

The last 4 bars of the figure show the shared IBTC and sieve with return cache and RATS mechanisms. Note that the relative position of the sieve and shared IBTC bars remains approximately the same for both the return cache and RATS. The figure shows, though, that the table size no longer dominates the total size of the techniques. The table size for the RATS is only 12K bytes compared to the return cache's 128K bytes, yet the RATS always has a higher memory consumption. This trend can be explained by the fact that the translation for the RATS is significantly larger than for the return cache. Further, both the return cache and the RATS need additions to the translations of call instructions (both direct and indirect). Consequently, both techniques are more closely tied to the number of unique call and return instructions that are translated than other techniques. Ultimately the longer translations for the RATS hurt both performance and memory consumption compared to other techniques.

The results in this section demonstrate that the performance benefit from a particular method of handling IBs varies with the architecture and program. There is no single best mechanism and careful attention must be paid to the selection of the mechanism to get good performance. In fact, simply moving from the best data cache hashing mechanism to a carefully crafted instruction cache hashing scheme reduces SPEC INT2000 average overhead significantly; over 50% of the overhead is removed (19% to 9%) on the Xeon processor. Likewise, the overall SPEC overhead on the Opteron can be reduced by 50% (4% to 2%). With careful selection of the mechanism, the average overhead of a SDT system for the SPEC benchmarks can be reduced to just 3.5% on the UltraSPARC-IIi, 4.5% on the Xeon, and 2.2% on the Opteron (Figure 26).

To verify that our results are broadly applicable to modern, object-oriented benchmarks, we also verified our results with the SPEC CPU2006 benchmark suite. We found no significant difference in the mechanism or configuration that should be used

Table II. Recommendations for Configuration Choices

Mechanism	Choice	Description
IBTC	Table size	8K entries does well, 32K entries gets maximum benefit
	Lookup placeup	code inside fragment
	Reprobing	Use reprobing with small IBTC tables (e.g., 1K) when memory constrained
	Adaptively resizing	Shared, fixed 32K-entry table over non-shared, adaptive
	Sharing	Shared, fixed 32K-entry table over non-shared adaptive
Sieve	Size	Size based on generated lookup code. SPARC: use a 1K-entry sieve to avoid extra `and` instruction, Pentium and Opteron: use a 16K-entry sieve to avoid `pushf` instruction
Inline	Inline amount	Use from zero to three inline translations
	Target selection	For call-type indirects, use a naive policy (inline two entires). For switch-type indirects, use profile guided policy
	Profiling	For switch-type indirects, use online profiling with a threshold of 30 executions
	Indirect type	Distinguish call-type from switch-type to handle efficiently
	Fall back	Select based on target architecture
Return Cache	Size	4K entries avoid conflicts, but larger may be needed to avoid extra instructions. Use only when ISA supports 32-bit immediate and immediate addressing
Fast Returns		Use when viable

for these benchmarks. Overall overhead for the SPEC CPU2006 benchmarks on the Opteron 244 machine were 13% and 5% when using the 16K-entry sieve with the 16K-entry return cache and fast returns, respectively. One notable difference is that the fast return mechanism is unsuitable for two benchmarks (`471.omnetpp` and `453.povray`) due to nonstandard use of the application's return address (for exception handling purposes).

Our findings indicate that SDT can have extremely low overhead for many benchmarks. These low overheads make SDT a feasible and beneficial technology in many settings. Since modern machines have gigabytes of memory, we make final recommendations based on performance alone. Table II summarizes the recommended configuration options.

6. RELATED WORK

Dynamic binary translation is a popular area of study, with a large design space, resulting in much research exploring design trade-offs. Trace layout [Duesterwald and Bala 2000; Hiniker et al. 2005], code cache management [Hazelwood and Smith 2003; 2004; Zhou et al. 2005; Kumar et al. 2005], and transparency [Bruening and Amarasinghe 2005] have all been studied in detail. Much work has been also been done investigating how SDT schemes can handle IBs, and a good overview of these options is given by Smith and Nair [2005]. Part of the motivation of this work is the fact that there are many options for handling IBs, but it is very difficult to compare any two techniques directly. In practice, most systems have chosen a single technique for handling IBs and therefore are unable to directly compare their technique to other techniques that have been developed.

Bruening [2004] and Bruening et al. [2003] detail numerous general design choices for handling IBs, including transparency issues with using the stack for scratch storage and choosing proper hashes to minimize data cache pressure. He also looks at IA-32-specific issues including a method to do comparisons on the IA-32 without altering the `eflags`, and also techniques for minimizing the cost of saving `eflags` when it is required.

Both Dynamo [Bala et al. 2000] and Daisy [Ebcioğlu and Altman 1997; Ebcioğlu et al. 2001] use chains of inlined comparisons to handle IBs. Pin uses a technique similar to

our IBTC with inlining [Luk et al. 2005]. The sieve was first introduced by HDTrans as a method for doing IB handling without polluting the data cache [Sridhar et al. 2005]. These are all pure software techniques for handling IBs. Kim and Smith [2003] have examined hardware techniques.

7. CONCLUSIONS

Software Dynamic Translation (SDT) is a powerful technique in which a running binary is dynamically modified to provide a variety of benefits. Binaries can be dynamically translated to new systems, protected from malicious intrusion, dynamically optimized, or instrumented to collect a variety of statistics. One major deterrent to more pervasive use of SDT is the overhead associated executing the program within an SDT system.

Handling IBs efficiently has been shown to be extremely important to having an efficient dynamic translation system. Furthermore, a variety of techniques have been proposed, but a thorough, cross-platform comparison of techniques has been lacking. This work addresses that issue by fully describing, implementing, and evaluating a variety of IB handling mechanisms. We use a publicly available, retargetable SDT system, three common architectures, and the full suite of SPEC CPU2000 benchmarks. Our findings indicate that fast returns are a viable option for a variety of benchmarks and provide significant runtime overhead reduction. When they are not viable, the best option for handling return instructions depends highly on the architectural support for 32-bit constants.

Our findings also indicate that for nonreturn instructions, moderately sized hashes provide most of the runtime benefit. Furthermore, placement of the code to perform the hash lookup (inline and duplicated, versus out of line with extra instructions) is of little importance. We do find, however, that data cache hashing is the most useful technique across platforms. What is more important, however, is choosing a set of instructions that is efficient on the target machine. For example, the saving and restore of the `eflags` on the Pentium 4 Xeon and Opteron 244 is so costly that avoiding the save and restore of the flags is significantly more important than instruction cache or data cache handling. Lastly, we find that a novel approach to using an inline cache entry can provide a performance benefit based on the underlying processor organization, removing as much as 50% of the overhead (4% to 2% for all of the SPEC benchmark suite) on an AMD Opteron.

REFERENCES

Advanced Micro Devices. 2006. AMD website on Opterons. `http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8826,0%0.html`.

Apple Computers. 2006. Apple website on Rosetta. `http://www.apple.com/rosetta/`.

Bala, V., Duesterwald, E., and Banerjia, S. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*. ACM Press, New York, 1–12.

Baraz, L., Devor, T., Etzion, O., Goldenberg, S., Skaletsky, A., Wang, Y., and Zemach, Y. 2003. IA-32 execution layer: A two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 191.

Bruening, D. 2004. Efficient, transparent, and comprehensive runtime code manipulation. Ph.D. thesis, MIT.

Bruening, D. and Amarasinghe, S. 2005. Maintaining consistency and bounding capacity of software code caches. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)*. IEEE Computer Society, Los Alamitos, CA, 74–85.

Bruening, D., Garnett, T., and Amarasinghe, S. 2003. An infrastructure for adaptive dynamic optimization. In *Proceedings of the 1st International Symposium on Code Generation and Optimization*. 265–275.

Chen, W.-K., Lerner, S., Chaiken, R., and Gillies, D. 2000. Mojo: A dynamic optimization system. In *Proceedings of the ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*.

CHERNOFF, A., HERDEG, M., HOOKWAY, R., REEVE, C., RUBIN, N., TYE, T., YADAVALLI, S. B., AND YATES, J. 1998. FX!32: A profile-directed binary translator. *IEEE Micro 18,* 2, 56–64.

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

DITZEL, D. R. 2000. Transmeta's Crusoe: Cool chips for mobile computing. In *Hot Chips XII. Stanford University.* IEEE Computer Society Press.

DUESTERWALD, E. AND BALA, V. 2000. Software profiling for hot path prediction: Less is more. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*. ACM Press, New York, 202–211.

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

EBCIOĞLU, K., ALTMAN, E., GSCHWIND, M., AND SATHAYE, S. 2001. Dynamic binary translation and optimization. *IEEE Trans. Comput. 50,* 6, 529–548.

GSCHWIND, M., ALTMAN, E. R., SATHAYE, S., LEDAK, P., AND APPENZELLER, D. 2000. Dynamic and transparent binary translation. *Comput. 33,* 3, 54–59.

HAZELWOOD, K. AND KLAUSER, A. 2006. A dynamic binary instrumentation engine for the ARM architecture. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06)*. ACM, New York, 261–270.

HAZELWOOD, K. AND SMITH, J. E. 2004. Exploring code cache eviction granularities in dynamic optimization systems. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. IEEE Computer Society, Los Alamitos, CA, 89.

HAZELWOOD, K. AND SMITH, M. D. 2003. Generational cache management of code traces in dynamic optimization systems. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'36)*. IEEE Computer Society, Los Alamitos, CA, 169.

HINIKER, D., HAZELWOOD, K., AND SMITH, M. D. 2005. Improving region selection in dynamic optimization systems. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'38)*. IEEE Computer Society, Los Alamitos, CA, 141–154.

HU, W., WILLIAMS, D., DAVIDSON, J. W., HISER, J. D., KNIGHT, J. C., AND NGUYEN-TUONG, A. 2009. Security through diversity: Leveraging virtual machine technology. *IEEE Secu. Priv. 7,* 1, Special Issue on IT Monoculture 26–33.

Intel 2005. *IA-32 Intel Architecture Optimization Reference Manual*.

KIM, H.-S. AND SMITH, J. E. 2003. Hardware support for control transfers in code caches. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'36)*. IEEE Computer Society, Los Alamitos, CA, 253.

KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. 2002. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*.

KUMAR, N., BRUCE R, C., WILLIAMS, D., DAVIDSON, J., AND SOFFA, M. 2005. Compile-Time planning for overhead reduction in software dynamic translators. *Int. J. of Parall. Program. 33,* 2, 103–114.

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

MOORE, R. W., BAIOCCHI, J. A., CHILDERS, B. R., DAVIDSON, J. W., AND HISER, J. D. 2009. Addressing the challenges of dbt for the arm architecture. In *Proceedings of the ACM Conference on Languages Compilers and Tools for Embedded Systems (LCTES'09)*.

SCOTT, K. AND DAVIDSON, J. 2001a. Low-Overhead software dynamic translation. Tech. CS-2001-18. July.

SCOTT, K. AND DAVIDSON, J. 2001b. Strata: A software dynamic translation infrastructure. In *Proceedings of the IEEE Workshop on Binary Translation*.

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

SEDGEWICK, R. 1983. *Algorithms*. Addison-Wesley.

SKADRON, K., AHUJA, P., MARTONOSI, M., AND CLARK, D. 1998. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'31)*. 259–271.

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

SRIDHAR, S., SHAPIRO, J. S., AND BUNGALE, P. P. 2005. HDTrans: A low-overhead dynamic translator. In *Proceedings of the Workshop on Binary Instrumentation and Applications*. IEEE Computer Society.

STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC CPU2000 Benchmarks. http://www.specbench.org/osg/cpu2000.

Sun Microsystems 1997. *UltraSPARC-IIi User's Manual*. Sun Microsystems.

TRANSITIVE CORPORATION LTD. 2006. Transitive website. http://www.transitive.com/.

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

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

ZHENG, C. AND THOMPSON, C. 2000. PA-RISC to IA-64: Transparent execution, no recompilation. *IEEE Comput. 33,* 3, 47–52.

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

Received May 2009; revised September 2010; accepted February 2011