Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems

Jason D. Hiser, Daniel Williams
Wei Hu, Jack W. Davidson
Department of Computer Science
University of Virginia
{hiser,dww4s,wh5a,jwd}@cs.virginia.edu

Jason Mars, Bruce R. Childers
Department of Computer Science
University of Pittsburgh
jom5x@cs.virginia.edu, childers@cs.pitt.edu

Abstract

Software Dynamic Translation (SDT) systems are used for program instrumentation, dynamic optimization, security, intrusion detection, and many other uses. As noted by many researchers, a major source of SDT overhead is the execution of code which is needed to translate an indirect branch's target address into the address of the translated destination block.

This paper discusses the sources of indirect branch (IB) overhead in SDT systems and evaluates several techniques for overhead reduction. Measurements using SPEC CPU2000 show that the appropriate choice and configuration of IB translation mechanisms can significantly reduce the IB handling overhead. In addition, cross-architecture evaluation of IB handling mechanisms reveals that the most efficient implementation and configuration can be highly dependent on the implementation of the underlying architecture.

1. Introduction

Software dynamic translation (SDT) is a technology that enables software malleability and adaptivity at the instruction level by providing facilities for run-time monitoring and code modification. Many useful systems have been built that apply SDT, 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 [7, 19]. 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 [9], UQDBT that dynamically translates Intel IA-32 binaries to run on SPARC processors [30], and DAISY that translates PowerPC instructions to VLIW instructions [11]. Computer architecture tools like Shade and Embra use SDT to implement high-performance simulators [6], while Mojo and Dynamo dynamically optimize native binaries to improve performance [1, 5]. Recently, SDT has been used to ensure the safe execution of untrusted binaries [17, 21, 22, 24].

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 a SDT system. The mediation of program execution adds overhead, possibly in the form of time, memory size, disk space, or network traffic. For a 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 shows 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. Inspection of the graphs show 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.

To address this problem, this work evaluates methods for efficiently handling IBs in SDT systems. This paper 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 novel improvement on standard inline cache entries 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; and
- Experimental evidence that the best method for handling IBs depends on the features of the target architec-

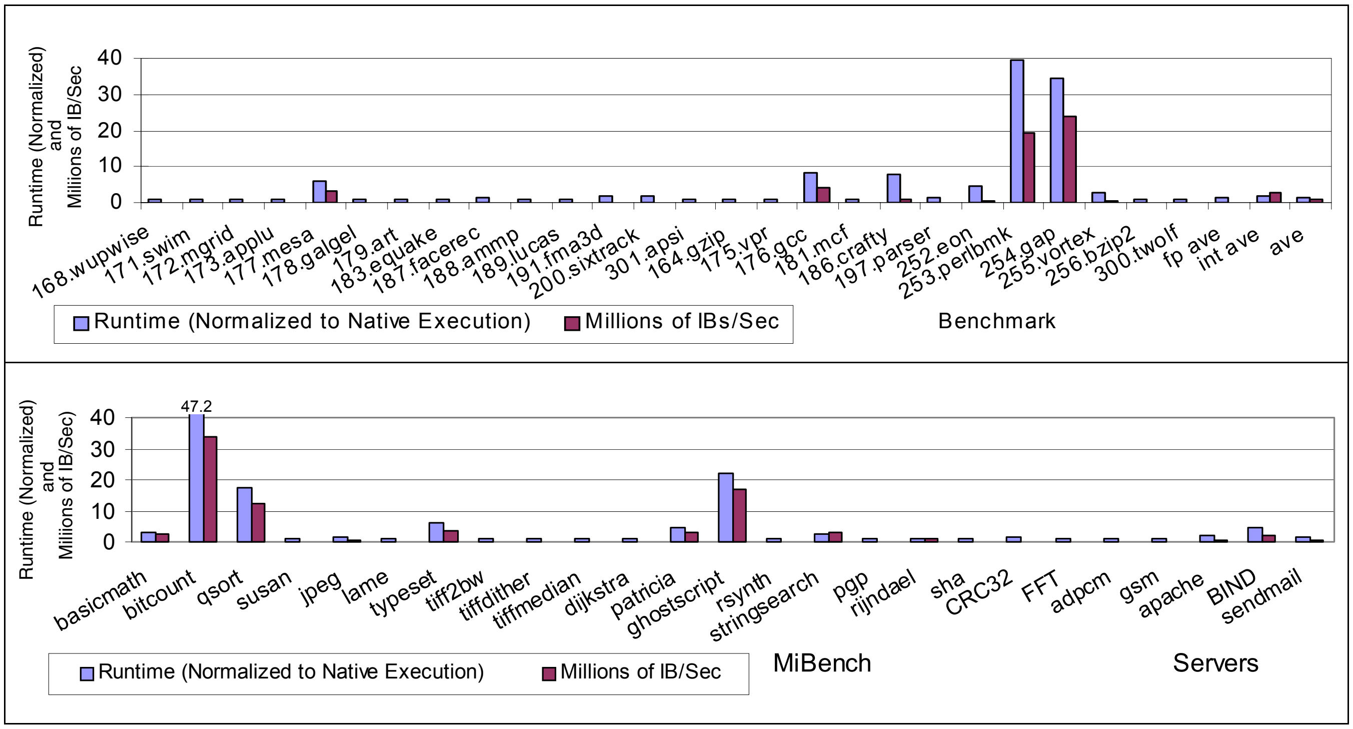


Figure 1: Overhead of SDT with naïve IB translation.

ture such as addressing modes, branch predictors, cache sizes and the ability to efficiently preserve architecture state.

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 paper is organized as follow. Section 2 gives a brief overview of SDT and Section 3 has a detailed description of the IB handling mechanisms evaluated in this paper. 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 [4, 20, 21].

Most dynamic translators operate as a co-routine with the application they are controlling. 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 [2, 21].

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.

Translation of direct branches is simple because there is only one branch target address. For IBs, the branch target address is computed at execution time which necessitates that the SDT system generate efficient code to perform the mapping at execution time.

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).

3.1. Indirect Branch Translation Cache

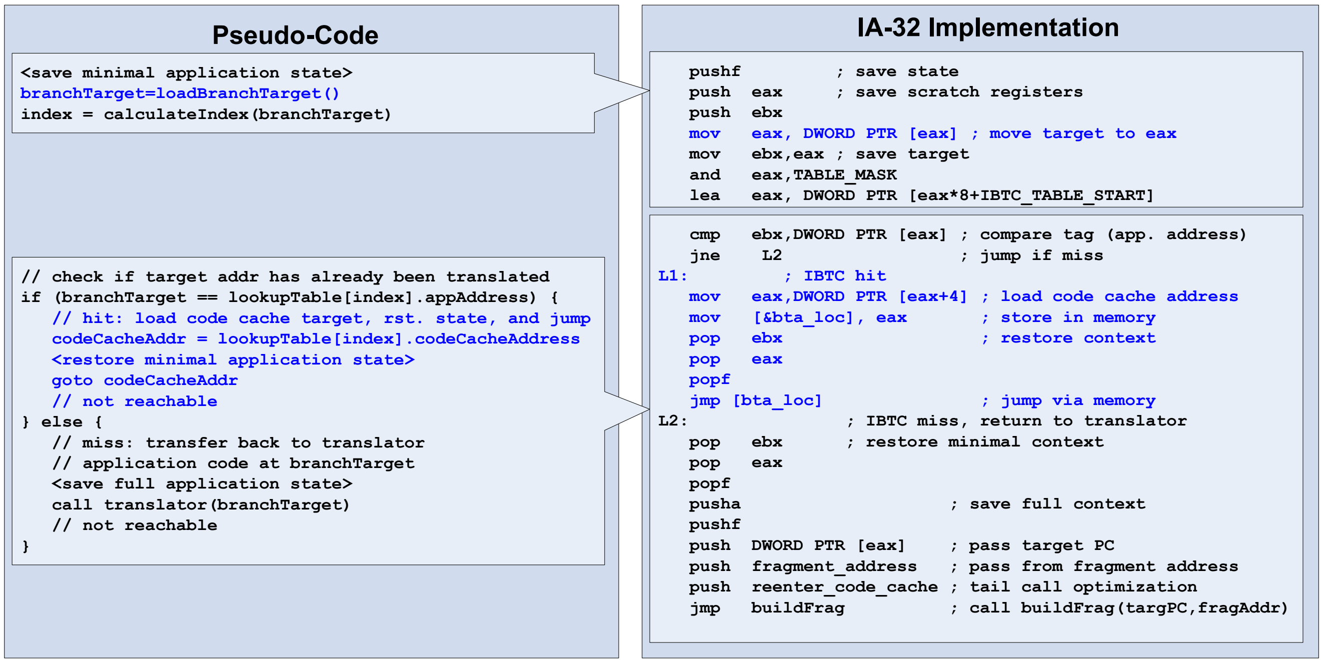
An Indirect Branch Translation Cache (IBTC) is a data structure used to translate the target of an IB to the corresponding code cache address [23]. An IBTC is a list of pairs of addresses—an application address and its corresponding code cache address. The left side of Figure 2 shows pseudo-code for performing an IB translation using an IBTC, and the right side of Figure 2 gives the 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 (non-shared). 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 non-shared, adaptive 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.



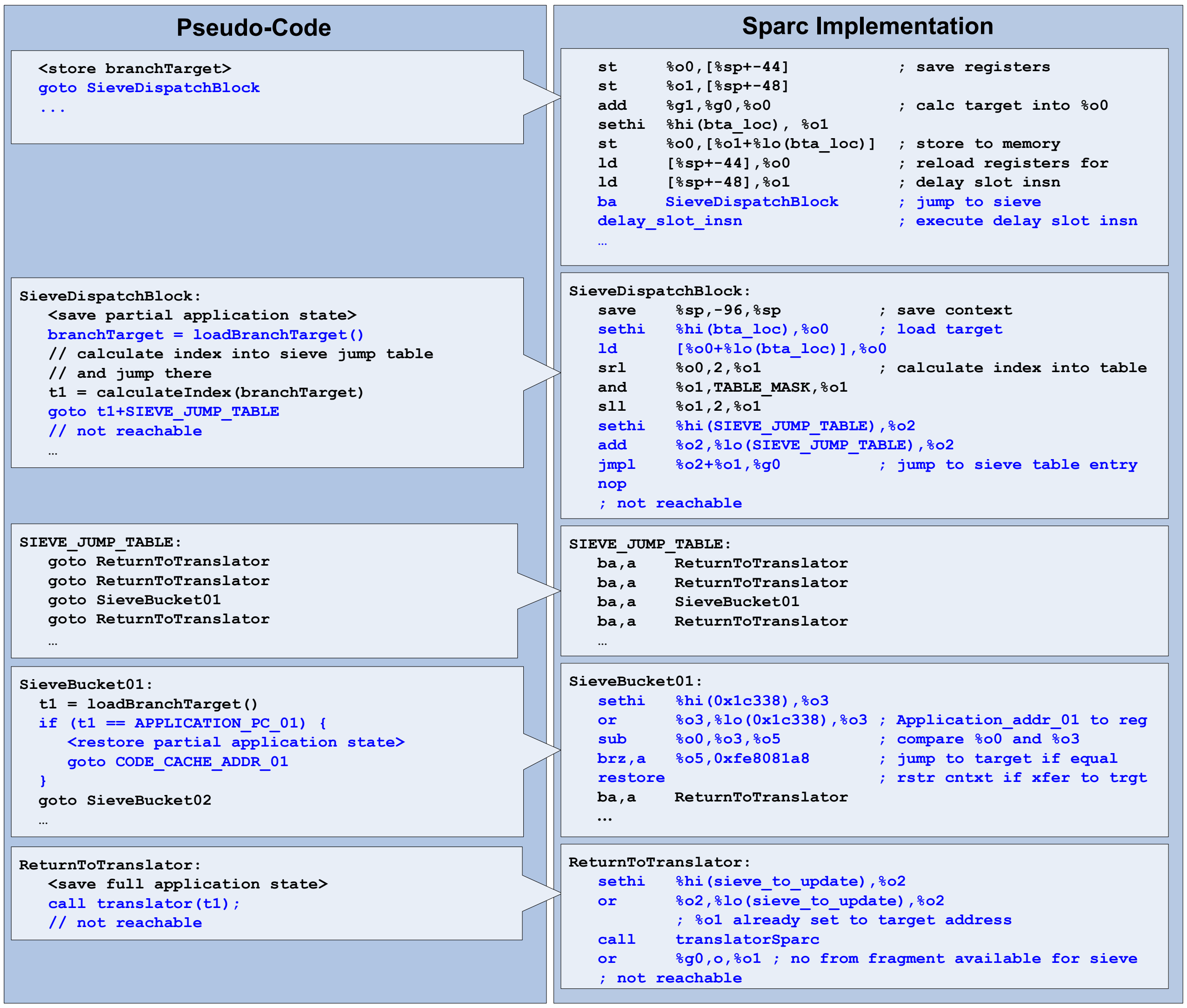
```
<save minimal application state>
branchTarget=loadBranchTarget()
index = calculateIndex(branchTarget)

// check if target addr has already been translated
if (branchTarget == lookupTable[index].appAddress) {
   // hit: load code cache target, rst. state, and jump
   codeCacheAddr = lookupTable[index].codeCacheAddress
   <restore minimal application state>
   goto codeCacheAddr
   // not reachable
} else {
   // miss: transfer back to translator
   // application code at branchTarget
   <save full application state>
   call translator(branchTarget)
   // not reachable
}
```

```
    pushf          ; save state
    push  eax      ; save scratch registers
    push  ebx
    mov   eax, DWORD PTR [eax] ; move target to eax
    mov   ebx,eax ; save target
    and   eax,TABLE_MASK
    lea   eax, DWORD PTR [eax*8+IBTC_TABLE_START]

    cmp   ebx,DWORD PTR [eax] ; compare tag (app. address)
    jne    L2                 ; jump if miss
L1:          ; IBTC hit
    mov   eax,DWORD PTR [eax+4] ; load code cache address
    mov   [&bta_loc], eax       ; store in memory
    pop   ebx                   ; restore context
    pop   eax
    popf
    jmp [bta_loc]               ; jump via memory
L2:                ; IBTC miss, return to translator
    pop   ebx      ; restore minimal context
    pop   eax
    popf
    pusha                      ; save full context
    pushf
    push  DWORD PTR [eax]     ; pass target PC
    push  fragment_address    ; pass from fragment address
    push  reenter_code_cache ; tail call optimization
    jmp   buildFrag          ; call buildFrag(targPC,fragAddr)
```

Figure 2: IBTC lookup algorithm and corresponding implementation for the IA-32.



Pseudo-Code

```
  <store branchTarget>
  goto SieveDispatchBlock
  ...
```

```
SieveDispatchBlock:
   <save partial application state>
   branchTarget = loadBranchTarget()
   // calculate index into sieve jump table
   // and jump there
   t1 = calculateIndex(branchTarget)
   goto t1+SIEVE_JUMP_TABLE
   // not reachable
   …
```

```
SIEVE_JUMP_TABLE:
   goto ReturnToTranslator
   goto ReturnToTranslator
   goto SieveBucket01
   goto ReturnToTranslator
   …
```

```
SieveBucket01:
  t1 = loadBranchTarget()
  if (t1 == APPLICATION_PC_01) {
     <restore partial application state>
     goto CODE_CACHE_ADDR_01
  }
  goto SieveBucket02
  …
```

```
ReturnToTranslator:
   <save full application state>
   call translator(t1);
   // not reachable
```

Sparc Implementation

```
   st     %o0,[%sp+-44]          ; save registers
   st     %o1,[%sp+-48]
   add    %g1,%g0,%o0            ; calc target into %o0
   sethi  %hi(bta_loc), %o1
   st     %o0,[%o1+%lo(bta_loc)] ; store to memory
   ld     [%sp+-44],%o0          ; reload registers for
   ld     [%sp+-48],%o1          ; delay slot insn
   ba     SieveDispatchBlock     ; jump to sieve
   delay_slot_insn               ; execute delay slot insn
   …
```

```
SieveDispatchBlock:
   save    %sp,-96,%sp          ; save context
   sethi   %hi(bta_loc),%o0     ; load target
   ld      [%o0+%lo(bta_loc)],%o0
   srl     %o0,2,%o1            ; calculate index into table
   and     %o1,TABLE_MASK,%o1
   sll     %o1,2,%o1
   sethi   %hi(SIEVE_JUMP_TABLE),%o2
   add     %o2,%lo(SIEVE_JUMP_TABLE),%o2
   jmpl    %o2+%o1,%g0          ; jump to sieve table entry
   nop
   ; not reachable
```

```
SIEVE_JUMP_TABLE:
   ba,a    ReturnToTranslator
   ba,a    ReturnToTranslator
   ba,a    SieveBucket01
   ba,a    ReturnToTranslator
   …
```

```
SieveBucket01:
   sethi   %hi(0x1c338),%o3
   or      %o3,%lo(0x1c338),%o3 ; Application_addr_01 to reg
   sub     %o0,%o3,%o5          ; compare %o0 and %o3
   brz,a   %o5,0xfe8081a8       ; jump to target if equal
   restore                      ; rstr cntxt if xfer to trgt
   ba,a    ReturnToTranslator
   …
```

```
ReturnToTranslator:
   sethi   %hi(sieve_to_update),%o2
   or      %o2,%lo(sieve_to_update),%o2
           ; %o1 already set to target address
   call    translatorSparc
   or      %g0,o,%o1 ; no from fragment available for sieve
   ; not reachable
```

Figure 3: Sieve algorithm and corresponding implementation for the SPARC.

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 [27]. The sieve is essentially an open hashing technique implemented solely with instructions [25]. The left side of Figure 3 gives the pseudo-code for an implementation of the translation code that a dynamic translator would generate and place in the code cache. The right side of Figure 3 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 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 return to translator 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 pseudo-code presented, the third entry in the sieve jump table has been modified to jump to `SieveBucket01`.

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 aspect to note is that the sieve never compares two arbitrary values. Instead, the sieve repeatedly compares one arbitrary value to a constant. Additionally, only one temporary is needed, so the overhead of the context save and restore may be lower. 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 code to do a tag compare and transfer on a match using only immediate values held within the instruction sequence itself. 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 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 reduce cache pollution.

The pseudocode for IB inlining is shown in Figure 4. `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.

```
1.   <save minimal application state>
2.   t1 = branchTarget
3.   if(t1 == STORED_TARGET) {
4.      <restore minimal application state>
5.      goto CODE_CACHE_ADDRESS
6.   }
```

Figure 4: Pseudo-code for IB inlining.

There are a number of parameters to consider when using inline cache entries. One question is 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? Do call-type IBs need to be treated separately from switch-type IBs? Section 5.3 empirically evaluates these options.

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 [21]. Sections 4.1–4.3 discuss our assumptions, machines, compilers, compiler options, and benchmarks we used for evaluation purposes.

4.1. Return Instructions

Return instructions are a special kind of IB that SDT systems can handle efficiently. Instead of emitting a complex sequence of instructions to handle returns like a generic IBs, returns are left untranslated and copied directly into the code cache. Normally leaving return instructions untranslated 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 text address. Control is maintained by translating call instructions (both direct and indirect) specially. Call instructions normally write their return address into a specified location (e.g., the stack or a special register). Calls are translated to write the corresponding code cache return address. Such translation causes the return instruction to return to another fragment within the code cache, and the translator retains control.

Since we have chosen to enable this *fast return* mechanism, return instructions do not undergo the normal IB translation and are effectively removed from consideration in our experiments. As return instructions are the most common type of IB, we believe this to be the best choice. The predictable nature of return instructions makes a variety of return-specific optimizations possible, such as a shadow stack or a *return cache* as presented by Sridhar [27]. Thus, we believe return instructions should be handled separately and consider research about choosing the best translation mechanism for returns to be separate from the work presented here.

4.2. Machines

The techniques presented in Section 3 were evaluated on three platforms—an Intel Pentium IV Xeon [15], a Sun UltraSPARC-IIi [29], and an AMD Opteron 244 [8]. The Pentium IV 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 IV 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.

4.3. Benchmarks

The full set of SPEC CPU2000 benchmarks was used for evaluation in this work [28]. However, as Figure 1 shows, SDT overhead is directly related to the rate of execution of IBs. Consequently, most graphs report results (the *SPEC number* for 3 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.

5. Experimental Results

This section evaluates the different IB handling techniques discussed in Section 3. Table 1 summarizes the configuration options for each IB handling mechanism.

5.1. IBTC

5.1.1. 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 5 shows the results for the Pentium IV 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.

5.1.2. Lookup code placement. Second, we determined if the placement of the IBTC lookup code effects performance. Figure 6 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 IV 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 AMD and Sun machines and the detailed results are omitted. Subsequent experiments evaluate IBTC options using inline IBTC lookup code.

5.1.3. Reprobing. Next, we considered the benefit of reprobing the IBTC to handle conflicts. Figure 7 shows the performance on the Pentium IV 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. Results on the UltraSPARC-IIi and Opteron 244 are similar and are omitted. Since we use an IBTC large enough to avoid most conflicts, the reprobing mechanism was not enabled for the subsequent IBTC experiments.

Table 1: 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 indirect branches or separate tables for each indirect branch
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

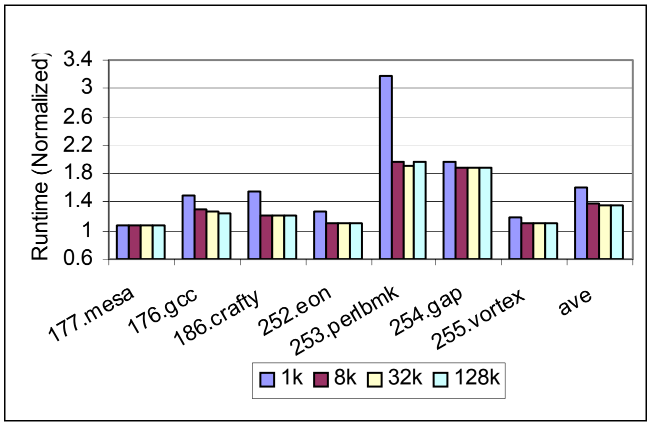


Figure 5: Performance with varying IBTC sizes (Pentium).

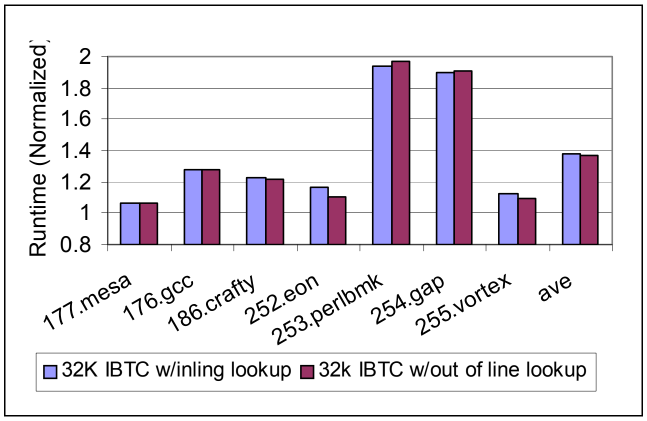


Figure 6: Placement of code to perform IBTC lookups (Pentium).

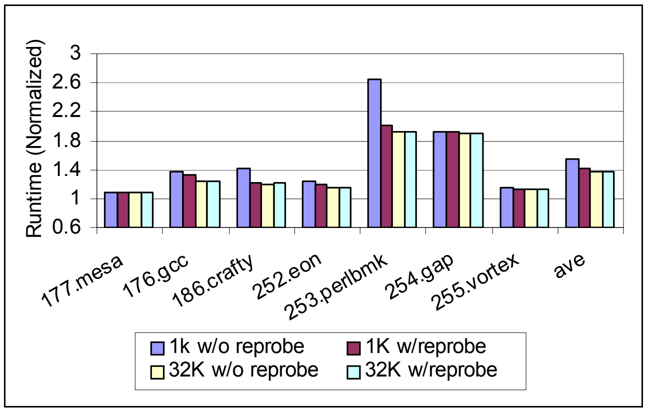


Figure 7: Performance of IBTC reprobing on conflict (Pentium).

5.1.4. 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 conflict misses by resizing an IBTC when a collision is detected on a cold miss. Figure 8 shows the results of assigning a small IBTC (8-entry) to each IB and doubling the size of the IBTC on any conflict for the Pentium IV architecture. The figure shows that adaptively resizing an IBTC adds little benefit over a large non-adaptive, shared IBTC.

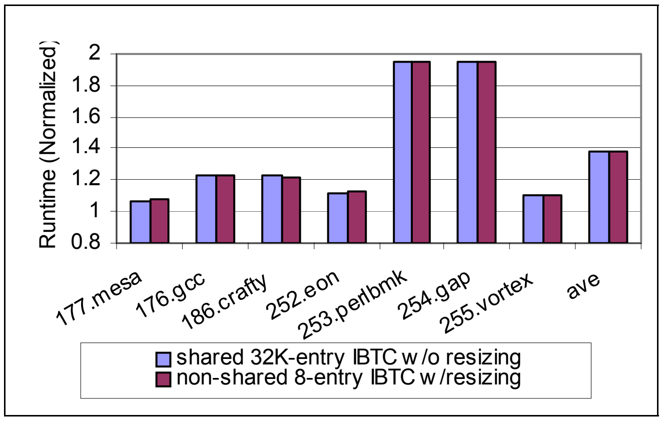


Figure 8: Performance with adaptive resizing of IBTC (Pentium).

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

5.2. Sieve

The sieve, as described in Section 3.2 and in previous work [27], has been implemented for all the machines we use for evaluation. We evaluated sieve sizes from 1 entry up to 128K entries. Figure 9 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` operation (see Figure 3) is 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 10 shows the results on the Pentium IV 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 by using the `MOVZWL` instruction (which does not affect the flags) instead of the `AND` instruction (which does write 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 Opteron

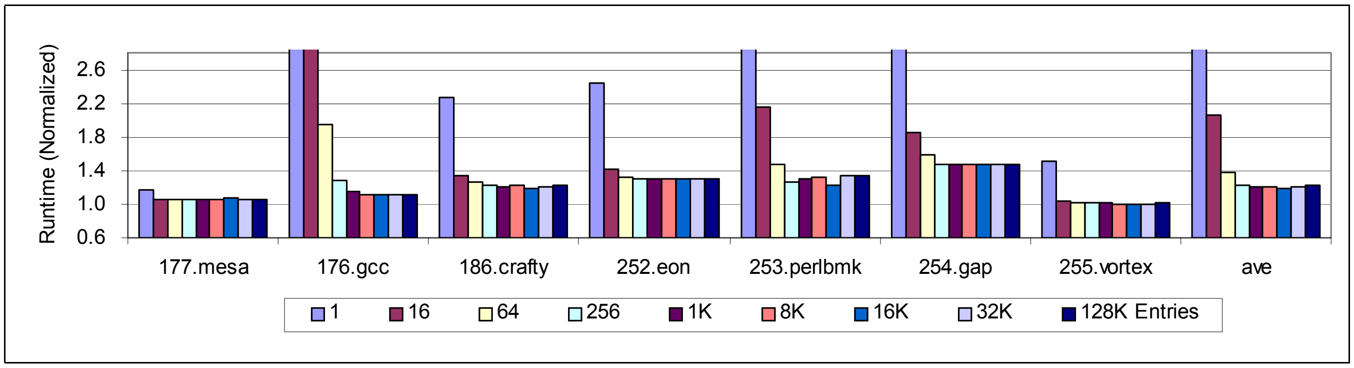


Figure 9: Sieve size experiment (UltraSPARC-IIi).

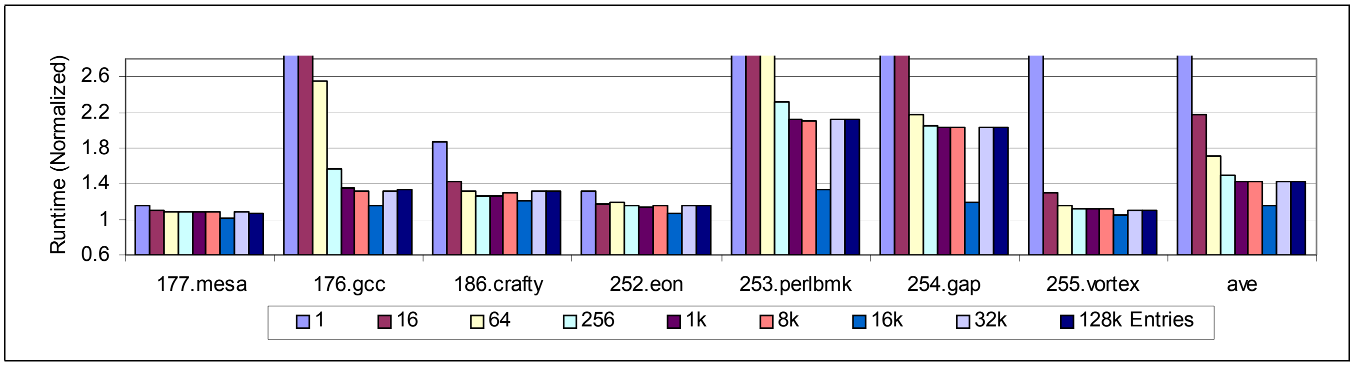


Figure 10: Sieve size experiment (Pentium).

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` operation, while a 16K-entry sieve is best on the Opteron and Xeon as it avoids the use `pushf` and `popf` instructions.

5.3. Inline Cache Entries

The next option is how to use inline cache entries. Figure 11 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 naively choosing the first target causes over 60% of all 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 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 11 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 is important. In the worst case, the last entry in the inline sequence might be hit. Yet, hitting in the last inline cache entry is expensive because several entries have to be traversed before getting to the last one.

Figure 12 shows the average number of instructions needed to satisfy an IB translation for inline depths of 1 to 5 for 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 com-

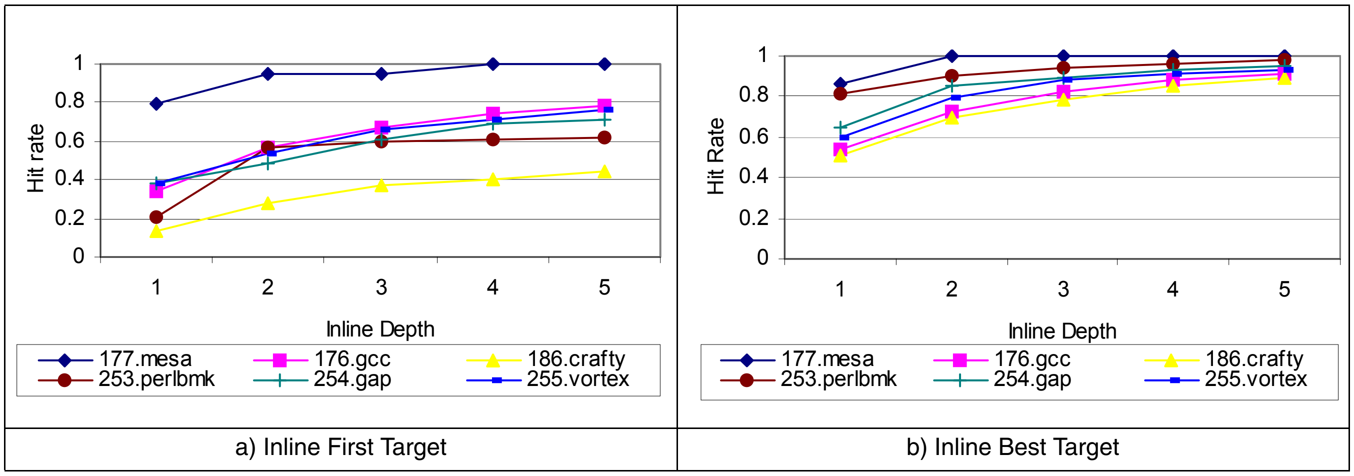


Figure 11: Inline hit rate for 1 to 5 entries (Opteron).

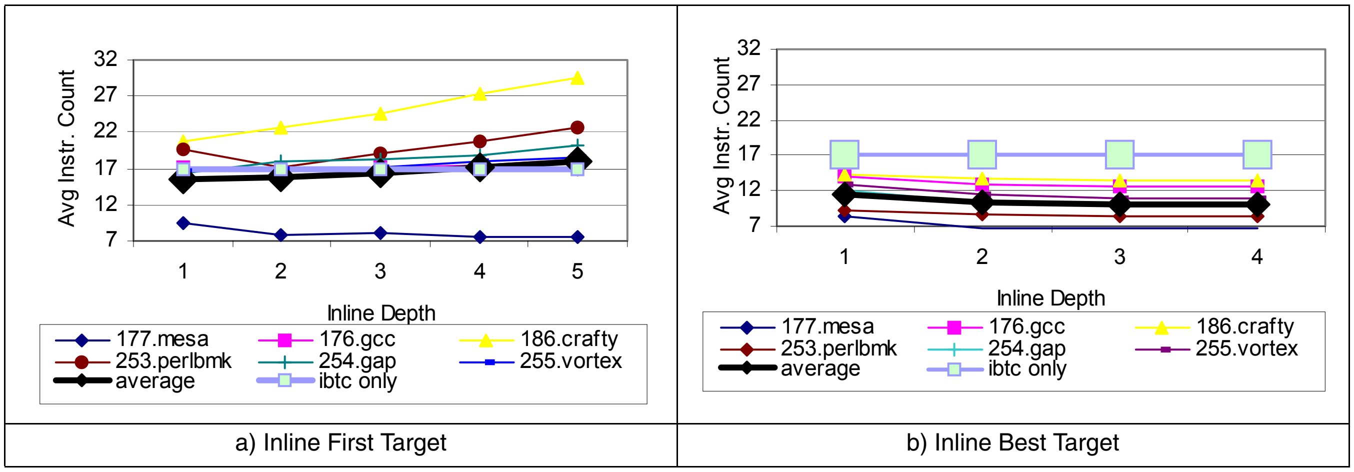


Figure 12: Average instruction count for 1 to 5 inline entries (Opteron).

pared 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 affect 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 11 and Figure 12, 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 entries once the profile selection is complete. Thus, if the profile indicates there are no dominate IB targets, the translator is free to re-write 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 13 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 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.

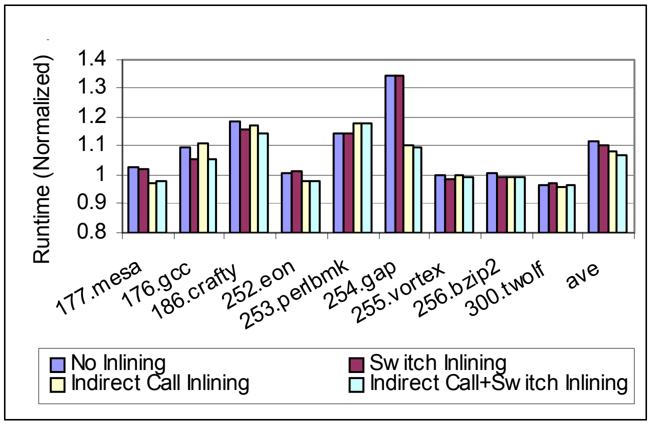


Figure 13: Inlining results for Opteron.

Even though the indirect call and switch inlining techniques are apparently independent, the combination of the two techniques frequently yields performance worse performance 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 to see significant benefits from using inline cache entries. 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, extra instructions are required to generate the 32-bit addresses required 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%. Unlike the Opteron, the Xeon does not see the benefits of inlining 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 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 2-5 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 frequent cache flushes. 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 of inlining. Call inlining should be performed using the first two dynamically taken targets, avoiding the profiling overhead.

5.4. Comparison

Now that we have explored the design and parameter space for IBTCs, the sieve, and branch target inlining, we examine which is most appropriate for each machine. For this evaluation, we compare the execution time of the benchmarks with different IB translation mechanisms. Other factors may be of importance in some systems, in particular memory consumption. However, on today's modern processors with gigabytes of RAM, even the largest IB handling mechanisms consume relatively little memory. Thus, due to space restrictions, we only consider performance.

Figure 14, Figure 15, and Figure 16 compare an IBTC to a sieve implementation on the UltraSPARC-IIi, Opteron 244, and Pentium IV Xeon, respectively. On the UltraSPARC-IIi, the sieve under performs the IBTC. The SPARC's limited addressing 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 equally well when the flags are saved. However, like the Xeon, once the save/restore of the flags is avoided the sieve performs much better.

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 SPECint2000 average overhead signifi-

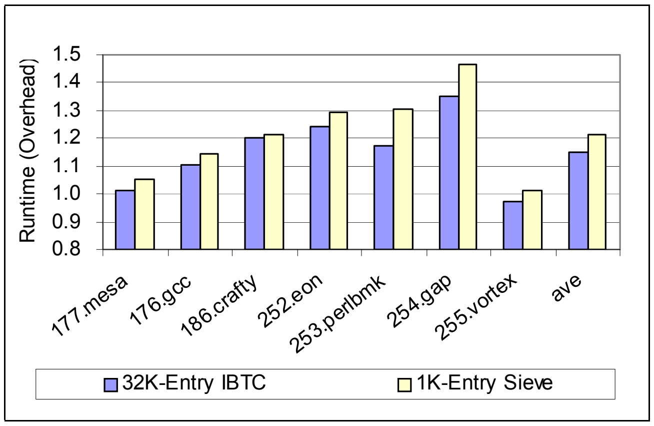


Figure 14: Sieve vs. IBTC on UltraSPARC-IIi.

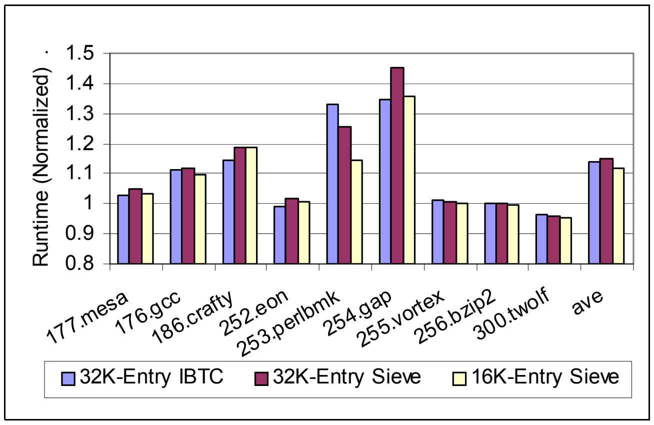


Figure 15: Sieve vs. IBTC on Opteron 244.

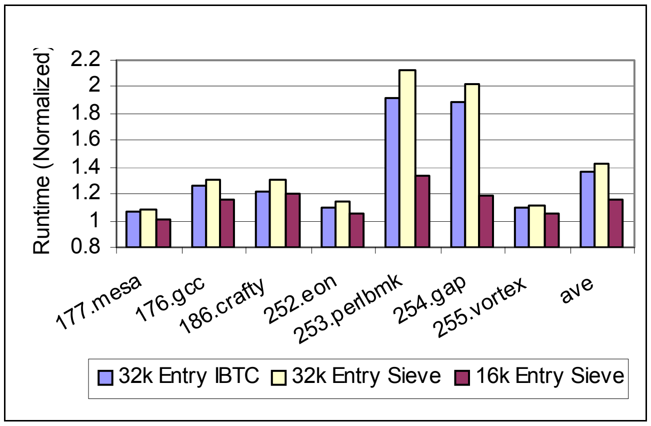


Figure 16: Sieve vs. IBTC on Pentium IV Xeon.

cantly; over 50% (19% to 9%) of the overhead is removed on the Xeon processor. Likewise, the overall SPEC overhead on the Opteron can be reduced by 50% (4% to 2% overhead). 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 17). These low overheads make SDT a feasible and beneficial technology in many settings. Finally, Table 2 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 [10, 14], code cache management [13, 12, 31, 18], and transparency [3] have all been studied in detail. Much work has been also been done investigating how SDT schemes can handle IBs, a good overview of these options is given by Smith and Nair [26]. Part of the motivation of this work is that there are many options when 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 details 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 [2, 4]. 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 [1] and Daisy [11] use chains of inlined comparisons to handle IBs. Pin uses a technique similar to our IBTC with inlining [20]. The sieve was first introduced by HDTrans as a method for doing IB handling without polluting the data cache [27]. These are all pure software techniques for handling IBs. Kim and Smith have examined hardware techniques [16].

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 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

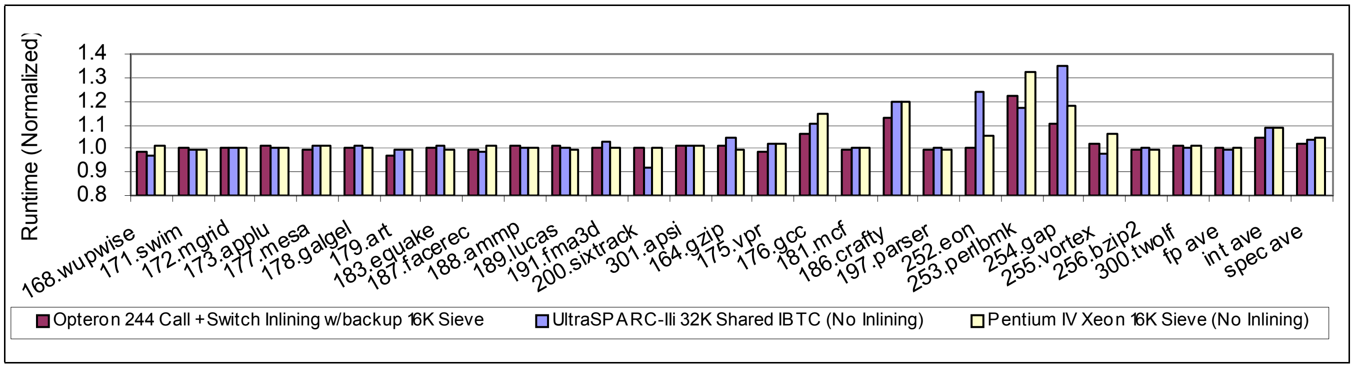


Figure 17: Performance with best configuration.

Table 2: Recommendations for configuration choices.

Mechanism	Choice	Description
IBTC	*Table size*	8k entries does well, 32k entries gets maximum benefit
	Lookup placement	Place lookup code inside fragment
	Reprobing	Use reprobing with small IBTC tables (e.g., 1K) when memory constrained
	Adaptively resizing	Shared, fixed 32K-entry table does better than non-shared, adaptive
	Sharing	Shared fixed 32k entry table does better than 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 indirect branches, use a naive policy (inline two entries) For switch-type indirect branches, use profile guided policy
	Profiling	For switch-type indirect branches, use online profiling with a threshold of 30 executions
	Indirect type	Distinquish call-type from switch-type to handle efficiently
	Fall back	Select based on target architecture

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 IV 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 benefits 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.

8. Acknowledgements

This research was support by the National Science Foundation under grants CNS–0305144, CNS–0524432, CNS–0551560, CNS–0305198, CNS-0551492, CNS-0509115, and CNS-0305198. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

9. References

[1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 1–12, New York, NY, USA, June 2000. ACM Press.

[2] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, 2004.

[3] Derek Bruening and Saman Amarasinghe. Maintaining consistency and bounding capacity of software code caches. In *CGO '05: Proceedings of the International Symposium on Code Generation and Optimization*, pages 74–85, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, pages 265–275, March 2003.

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

[6] Bob Cmelik and David Keppel. 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*, pages 128–137, New York, NY, USA, May 1994. ACM Press.

[7] Apple Computers. Apple website on rosetta, 2006.

[8] Advanced Micro Devices. AMD website on opterons, 2006.

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

[10] Evelyn Duesterwald and Vasanth Bala. 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*, pages 202–211, New York, NY, USA, 2000. ACM Press.

[11] Kemal Ebcioglu and Erik Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 26–37, New York, NY, USA, 1997. ACM Press.

[12] Kim Hazelwood and James E. Smith. Exploring code cache eviction granularities in dynamic optimization systems. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, page 89, Washington, DC, USA, 2004. IEEE Computer Society.

[13] Kim Hazelwood and Michael D. Smith. Generational cache management of code traces in dynamic optimization systems. In *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 169–179, Washington, DC, USA, 2003. IEEE Computer Society.

[14] David Hiniker, Kim Hazelwood, and Michael D. Smith. Improving region selection in dynamic optimization systems. In *MICRO 38: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 141–154, Washington, DC, USA, 2005. IEEE Computer Society.

[15] Intel. *IA-32 Intel Architecture Optimization Reference Manual*, 2005.

[16] Ho-Seop Kim and James E. Smith. Hardware support for control transfers in code caches. In *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 253, Washington, DC, USA, 2003. IEEE Computer Society.

[17] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, August 2002.

[18] Naveen Kumar, Bruce R. Childers, Daniel Williams, Jack W. Davidson, and Mary Lou Soffa. Compile-time planning for overhead reduction in software dynamic translators. *International Journal of Parallel Programming*, 33(2):103–114, 2005.

[19] Transitive Corporation Ltd. Transitive website, 2006.

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 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*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[21] Kevin Scott and Jack Davidson. Strata: A software dynamic translation infrastructure. In *IEEE Workshop on Binary Translation*, September 2001.

[22] Kevin Scott and Jack W. Davidson. Safe virtual execution using software dynamic translation. In *Proceedings of the 18th Annual Computer Security Applications Conference*, pages 209–218, Las Vegas, NV, December 2002.

[23] Kevin Scott, N. Kumar, Bruce Childers, Jack W. Davidson, and Mary Lou Soffa. Overhead reduction techniques for software dynamic translation. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, page 200. IEEE Computer Society, 2004.

[24] Kevin Scott, Naven Kumar, Siva Velusamy, Bruce Childers, Jack W. Davidson, and Mary Lou Soffa. Retargetable and reconfigurable software dynamic translation. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society.

[25] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1983.

[26] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

[27] Swaroop Sridhar, Jonathan S. Shapiro, and Prashanth P. Bungale. Hdtrans: A low-overhead dynamic translator. In *Proceedings of the 2005 Workshop on Binary Instrumentation and Applications*. IEEE Computer Society, September 2005.

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

[29] Sun Microsystems. *UltraSPARC-IIi User's Manual*, 1997. User's Manual.

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

[31] Shukang Zhou, Bruce R. Childers, and Mary Lou Soffa. 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*, pages 100–109, New York, NY, USA, 2005. ACM Press.