Reducing Pressure in Bounded DBT Code Caches

José A. Baiocchi, Bruce R. Childers
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, USA
{baiocchi,childers}@cs.pitt.edu

Jack W. Davidson, Jason D. Hiser
Department of Computer Science
University of Virginia
Charlottesville, Virginia 22904, USA
{jwd,hiser}@virginia.edu

ABSTRACT

Dynamic binary translators (DBT) have recently attracted much attention for embedded systems. The effective implementation of DBT in these systems is challenging due to tight constraints on memory and performance. A DBT uses a software-managed code cache to hold blocks of translated code. To minimize overhead, the code cache is usually large so blocks are translated once and never discarded. However, an embedded system may lack the resources for a large code cache. This constraint leads to significant slowdowns due to the retranslation of blocks prematurely discarded from a small code cache. This paper addresses the problem and shows how to impose a tight size bound on the code cache without performance loss. We show that about 70% of the code cache is consumed by instructions that the DBT introduces for its own purposes. Based on this observation, we propose novel techniques that reduce the amount of space required by DBT-injected code, leaving more room for actual application code and improving the miss ratio. We experimentally demonstrate that a bounded code cache can have performance on-par with an unbounded one.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-purpose and application-based systems—*Real-time and embedded systems*; D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers, Incremental compilers, Interpreters, Optimization, Run-time environments*

General Terms

Measurement, Performance, Design, Experimentation

Keywords

Dynamic Binary Translation, Code Generation, Footprint Reduction, System-on-Chip

1. INTRODUCTION

Dynamic Binary Translators (DBT) allow the modification of a running program for a specific purpose. Recent work has shown several uses of DBT for embedded systems, including instruction set translation [5], security [14, 16], power management [18], and software caching [15, 2], For instance, a DBT can improve performance in embedded systems with Flash memory because it provides a form of incremental loading that requires fewer accesses to the slow Flash device, rather than loading the whole executable into main memory (DRAM). Previous work with a DBT for this type of system has achieved a 1.9x to 2.2x speedup over native execution [2]. Despite these promising uses, the adoption of DBT for embedded systems has been limited, due to tight constraints on memory and performance that make the implementation of an efficient DBT a challenging task.

A DBT commonly uses a software-managed memory buffer (a "code cache") to hold translated application instructions. To ensure low runtime overhead in general-purpose systems, the code cache size is usually unbounded to let it grow large enough to hold an application's full translated code working set. Thus, application code is never evicted and there is no need for re-translation due to premature evictions. When the code cache is unbounded, a DBT's performance is partially a function of the number of *compulsory misses*. Past work showed that a general-purpose DBT with an unbounded code cache has an average performance overhead of just 2% to 4% over native execution on the SPEC benchmarks [12].

Unfortunately, a large, unbounded code cache can easily exceed the limited capacity of an embedded system's memory resources (e.g., if the system has only a small scratchpad memory). As a result, the performance of a dynamically translated program in such systems can be poor. The problem is that limiting the size of the code cache makes it unlikely that the translated code working set will fit. If it becomes necessary to obtain space for newly translated code and the code cache is full, some of the existing translated code must be evicted. When code is evicted, it becomes possible to need that code again, and *capacity misses* appear.

Past research for general-purpose systems has examined eviction schemes that effectively decide what code to keep in the code cache. Even with these techniques, the code cache size can still be hundreds of kilobytes to a few megabytes [10]. For embedded systems, compression and pinning can be used to keep needed code in the code cache [2]. While these approaches are important, they primarily address *what code to keep and when to discard it*. They do not tackle the *footprint of the translated code*, although some work considers it in a limited way [2, 7, 6]. We show in this paper, the code inserted by the DBT for its own purposes can easily account for 70% of the code in the code cache! As a result, it is important to aggressively minimize this "control code".

We investigate and develop techniques that *reduce the footprint of the translated code* for small, bounded code caches. With a smaller translated code footprint, there is less pressure on the code

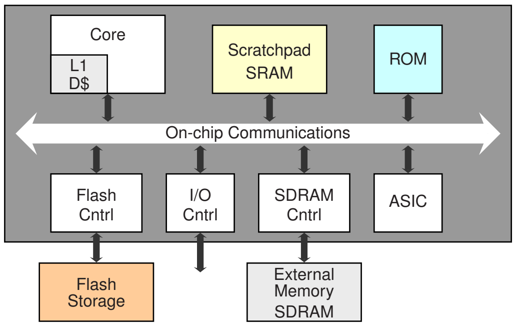


Figure 1: Example target embedded system

cache, and its miss ratio is improved. To this end, we categorize the instructions emitted by a DBT and distinguish those necessary to carry out the application behavior from those inserted by the DBT to ensure its control over execution. We measure the relative code cache space consumed by each category, and identify which aspects of the DBT's "control code" have the largest impact on code footprint. Then, we develop techniques to minimize it. A reduction in the code cache space consumed by "control code" leaves more room for actual application code, which lowers the number of evictions and achieves a significant improvement in performance.

This paper makes the following contributions:

- A classification of the instructions generated by a DBT, used to characterize the utilization of the code cache. This has not previously been done because in general-purpose systems there is less concern about code expansion.
- Experimental evidence of the excessive amount of code inserted by the DBT and its negative effect on performance under the memory constraints of embedded systems.
- Descriptions and implementations of novel techniques for substantially reducing the space consumed by DBT-injected code: single-instruction trampolines, factored indirect handling, prologue elimination, and bottom link eliding.
- Comprehensive experimental evaluation of the techniques, considering different code cache sizes and management approaches.

The paper is organized as follows: Section 2 provides the framework for our work. Section 3 shows the performance impact of reducing the size of the code cache and Section 4 analyzes the usage of the code cache, describes and incrementally evaluates different techniques for reducing the footprint of each instruction category, choosing the most beneficial. Section 5 gives the overall improvement with the selected techniques. Section 6 describes related work and Section 7 concludes.

2. FRAMEWORK

We start with a description of the type of embedded system targeted by our study and the operation of a typical DBT.

2.1 Target SoC System

Figure 1 shows a canonical embedded system that has a processor, L1 data cache (D-cache), an application-specific integrated circuit (ASIC), on-chip SRAM, a ROM, controllers for external Flash and main (SDRAM) memories and off-chip I/O channels. The on-chip SRAM is a scratchpad memory (SPM) that has single-cycle

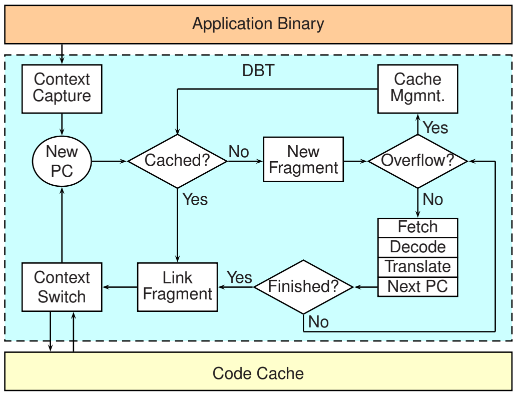


Figure 2: DBT system operation

access. The application code executes from SPM. The figure shows both SDRAM and Flash memories. The SDRAM is main memory and holds application data. The Flash memory is managed by the operating system; it holds user files, including application binary images. In this environment, the DBT is kept in ROM as part of the system code, and executed from there. The code cache is allocated to the SPM, which takes advantage of its fast (single-cycle) access time. This approach is essentially a form of software instruction caching [15, 2]. A hardware L1 instruction cache is unnecessary because code executes directly from the SPM.

2.2 Dynamic Binary Translation

Figure 2 shows a high-level view of a typical DBT. The DBT mediates application execution – it ensures that all untranslated application instructions are examined, and possibly modified, prior to their execution. As a result, the DBT must be invoked whenever new application code is requested. New application code is requested when a control transfer instruction (CTI) changes program flow to an application address that does not have a corresponding translated address in the code cache. To remain in control, the DBT rewrites CTIs to "re-enter" it when program flow goes to a new application address.

In the most basic mode of operation, the DBT is re-entered whenever a CTI is executed. To safely re-enter the translator, the application's context must be saved to free registers for use by the translator. In essence, a *context switch* is done to the DBT, which operates as a co-routine to the application. The translator is notified of the requested application address and checks whether translated code already exists for it. If so, the application context is restored and a jump is made to the translated code. Otherwise, the DBT builds a new *fragment* of translated instructions. A fragment is a code region that is translated as one unit. A hash table, the *fragment map*, records information about the new fragment, using the application address that corresponds to the fragment as a key. During translation, the code in the fragment is written to the code cache (also called a *fragment cache* (F$)). When the DBT finishes, the application context is restored and control is transferred to the translated code (in the F$).

The *Fragment Builder* is the DBT's component that fetches, classifies and translates instructions until a *stop condition* is met [11]. This condition indicates when to terminate a fragment. It depends on the type of instruction being processed and the DBT's region formation policy. For example, fragments could be terminated at all CTIs to form *dynamic basic blocks*.

Trampolines are portions of code emitted into the F$ to return control to the DBT for translating a new address. To reduce overhead from context switches, when a new fragment is created, all direct control transfers that would have requested its translation can be "linked" to it. That is, when the target code materializes in the F$, the trampolines are replaced by a direct control transfer to the target code [3]. The DBT's *Fragment Linker* is responsible for this task. It records trampolines and their target application addresses so they can be "fixed up" immediately after the pending application address is requested and translated. Indirect jumps and returns cannot be directly linked because their targets change as the program executes. In this case, to keep the execution in the F$, a separate mechanism, called the *Indirect Branch Target Cache* [12] is employed to map target application addresses to their corresponding translated addresses.

When a F$ is too small and cannot hold the full translated code working set of the program, it eventually overflows. *Code cache management* techniques are employed to handle this problem [3, 9, 10]. These solutions evict some or all translated blocks from the F$ to make room for new code, preemptively or on-demand. However, they are likely to increase the F$ miss rate because evicted code may be requested again and retranslated. The performance penalty due to retranslation can be high, especially in embedded systems where the binary image resides on slow Flash memory [2].

This paper investigates how the footprint of the translated code can be reduced so that it fits in the F$. With less pressure on the valuable F$ space, there will be fewer evictions and a better miss rate. Thus, less code will be re-fetched and re-translated, leading to better overall performance.

3. IMPACT OF MEMORY CONSTRAINT

To understand the effect of bounding the F$ to a small size, as in an embedded system with limited memory, we studied the performance of small fragment caches without our techniques to reduce code footprint. The F$ is constrained to the available SPM size.

3.1 Experimental Methodology

For this study, we used the Strata [17], a publicly available DBT. Strata was retargeted to SimpleScalar/PISA [1] and modified to include the techniques in this paper. Our experiments use the programs from MiBench [8] that SimpleScalar can execute[1], with the large input data sets.

We extended SimpleScalar's out-of-order simulator with Flash and SPM. We use the configuration shown in Table 1, which models the Marvell 624MHz XScale PXA-270 SoC, augmented with SPM, SDRAM and NOR Flash [13]. This processor is used in devices such as the Dell Axim Pocket PC. The Flash latencies were measured on a Dell Axim x50v with NOR Flash and a 8192-byte file buffer. It takes Windows Mobile Edition 5, 1.6 ms to fetch a block and 67,000 ns per word to read from the block. To facilitate experimentation, PISA uses a 64-bit instruction word; however, embedded processors use 16-bit or 32-bit instructions, so we double the SPM size for the simulations. We refer to the smaller *effective size* (e.g., a 32KB SPM is simulated with 64KB).

3.2 Performance of Small Fragment Caches

We study the effect of bounding the size of the F$ when running the MiBench programs under Strata's control. Strata was configured to always stop fragment formation when a control transfer instruction (CTI) is found, making the fragments *Dynamic Basic Blocks* (DBB). This configuration minimizes code duplication and is appropiate when memory is limited.

[1]`mad`, `ispell`, `rsynth`, and `sphinx` don't run due to library and system call incompatibility with SimpleScalar.

Processor (XScale PXA-270 624Mhz)			
`fetch:ifqsize`	8	`issue:width`	2
`fetch:mplat`	3	`res:ialu`	1
`fetch:speed`	1	`res:imult`	1
`bpred`	`bimod`	`res:fpalu`	1
`bpred:bimod`	128	`res:fpmult`	1
`bpred:btb`	512 4	`res:memport`	1
`decode:width`	1	`lsq:size`	4
`issue:inorder`	`true`	`ruu:size`	4
`issue:wrongpath`	`true`	`commit:width`	2
Memory System			
`tlb:dtlb`	`1:4096:32:f`	`scratchpad:lat`	1
`tlb:itlb`	`1:8192:32:f`	`tlb:lat`	30
`cache:il1`	`none`	`cache:dl1lat`	1
`cache:dl1`	`32:32:32:f`	`mem:width`	8
`cache:il2`	`none`	`mem:lat`	60 12
`cache:dl2`	`none`	`flash:lat`	900K 42K

Table 1: SimpleScalar Configuration

To avoid expensive context switches, Strata uses *fragment linking* and provides several techniques for handling indirects [12]. We chose a *shared IBTC* because it was determined to be the most useful technique across platforms [12]. PISA returns (`jr $ra`) are treated as indirects.

Performance results are reported for three F$ sizes normalized to the execution of a program under Strata's control with an unbounded F$. The unbounded F$ baseline represents the "ideal" performance if no constraint was placed on the F$ size – this configuration has no capacity misses and clearly illustrates the impact of a constraint.

Limiting the F$ size requires of code cache management to handle overflow events. We employed two techniques, *FLUSH* and *FIFO*, which are at opposite ends of the spectrum of eviction granularities and performance cost [10]. They illustrate the independence of our footprint reduction techniques from the eviction scheme. FLUSH is a low-overhead, coarse-grained technique that handles an overflow by evicting the whole contents of the code cache, which the DBT must translate again [3]. FIFO is a fine-grained technique that evicts only the least recently created fragment(s); it treats the code cache as a circular buffer [9]. FIFO has a better miss rate than FLUSH and avoids internal fragmentation.

Strata allocates fragments and trampolines interleaved in the F$. When a CTI is translated and its target is already in the F$, the generation of a trampoline can be avoided by emitting a CTI to the target fragment. This saves space with an unbounded F$ or when FLUSH is used. When using FIFO, space must be reserved to change the CTI into a trampoline when the target fragment is evicted.

Figure 3 shows the performance of the MiBench programs relative to the unbounded F$ baseline. For 64K and 32K F$ using FLUSH, some benchmarks (*adpcm.enc/dec*, *crc*, *blowfish.enc/dec*, *bitcount*) have similar performance as the unbounded baseline when the working set fits in the F$. However, with FIFO, all benchmarks have at least some slowdown. This slowdown is due to the effort of always generating trampolines and unlinking them on eviction, which increases both code size and frequency of evictions.

As the size of the F$ is reduced and F$ pressure is increased, performance suffers. Some benchmarks have considerable slowdowns with both FLUSH and FIFO. For instance, *fft* practically fits in a 64K F$; it has no slowdown with FLUSH and only 3% with FIFO. When running with a 32K F$, its slowdowns are 3.22x (FLUSH) and 12.9x (FIFO). However, a 16K F$ has slowdowns of 3418.97x (FLUSH) and 2846.71x (FIFO)! For *patricia*, signifi-

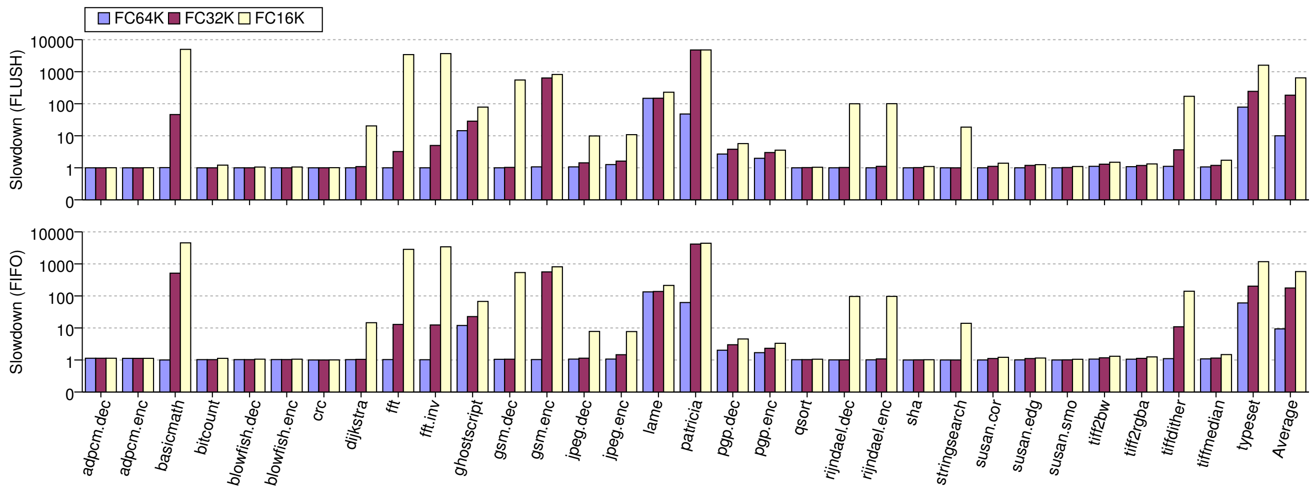


Figure 3: Initial performance relative to unbounded F$

cant slowdowns occur even with a 64K F$: 47.83x (FLUSH) and 62.25x (FIFO). The situation is especially bad for 16K: 4779.03x (FLUSH) and 4416.92x (FIFO)!

Different F$ sizes favor FIFO or FLUSH. *fft* does better with FLUSH in 32K F$, and better with FIFO in 16K F$. *patricia* prefers FLUSH for 64K F$ and FIFO for 32K and 16K F$. As F$ pressure increases, FIFO eventually performs better than FLUSH, but the inflexion point depends on the benchmark.

These results show that high F$ pressure leads to poor performance. Thus, we aim to reduce the size of the translated code so the pressure on the F$ is reduced. In this way, it can hold more fragments for a longer time before eviction. This approach is independent of the eviction policy – it will help either policy do better.

4. REDUCING TRANSLATED CODE SIZE

To find opportunities to reduce the translated code footprint, we analyzed the composition of the code emitted by the DBT into the F$. In general-purpose systems this type of analysis is rarely made, because there are looser constraints on F$ size and there is also performance benefit from code expansion when the F$ is unbounded [3, 11]. However, in embedded systems with limited code cache space, it is important to improve the utilization of the F$. The F$ should hold instructions that advance program execution rather than those inserted to transfer control back to the DBT.

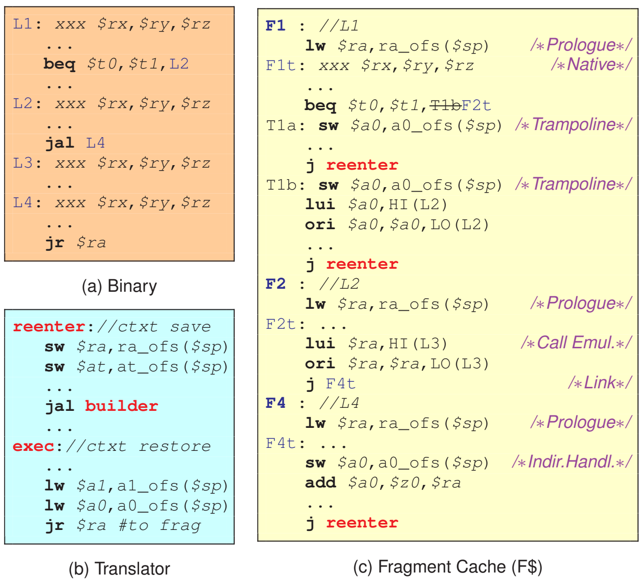
4.1 Instruction Categories

We classify the instructions put into the F$ according to their purpose. The instruction categories are illustrated in Figure 4, which shows an example of untranslated code (leftside) and its corresponding translated code (rightside).

Prologue instructions are executed to complete the context restore when returning from the DBT to the translated code. In PISA, the control transfer to the fragment is done with an indirect jump (see `exec` routine), which needs a free register. The register must be restored at the target fragment. All fragments in Figure 4 (`F1`, `F2`, `F4`) have the prologue: `lw $ra,ra_ofs($sp)`.

Native instructions are copied unmodified from the binary to the F$ or translated for some purpose. In Figure 4, fragment `F1` contains a series of native instructions (labelled "Native").

Trampoline instructions are used to return control to the DBT



```
L1: xxx $rx,$ry,$rz
    ...
    beq $t0,$t1,L2
L2: xxx $rx,$ry,$rz
    ...
    jal L4
L3: xxx $rx,$ry,$rz
    ...
L4: xxx $rx,$ry,$rz
    ...
    jr $ra
```

(a) Binary

```
reenter://ctxt save
    sw $ra,ra_ofs($sp)
    sw $at,at_ofs($sp)
    ...
    jal builder
    ...
exec://ctxt restore
    ...
    lw $a1,a1_ofs($sp)
    lw $a0,a0_ofs($sp)
    jr $ra #to frag
```

(b) Translator

```
F1 : //L1
    lw $ra,ra_ofs($sp)      /*Prologue*/
F1t: xxx $rx,$ry,$rz          /*Native*/
    ...
    beq $t0,$t1,~~T1b~~F2t
T1a: sw $a0,a0_ofs($sp) /*Trampoline*/
    ...
    j reenter
T1b: sw $a0,a0_ofs($sp) /*Trampoline*/
    lui $a0,HI(L2)
    ori $a0,$a0,LO(L2)
    ...
    j reenter
F2 : //L2
    lw $ra,ra_ofs($sp)      /*Prologue*/
F2t: ...
    lui $ra,HI(L3)         /*Call Emul.*/
    ori $ra,$ra,LO(L3)
    j F4t                        /*Link*/
F4 : //L4
    lw $ra,ra_ofs($sp)      /*Prologue*/
F4t: ...
    sw $a0,a0_ofs($sp)  /*Indir.Handl.*/
    add $a0,$z0,$ra
    ...
    j reenter
```

(c) Fragment Cache (F$)

Figure 4: Example fragments with instruction categories

when a CTI's target address is untranslated. Fragment `F1` has trampolines at `T1a` and `T1b`, corresponding to the branch's taken and not-taken application addresses, which are initially untranslated. After `F1` is executed, and the branch is taken, control returns to the DBT (see `reenter` routine). After the DBT creates the target fragment `F2`, the branch in `F1` is redirected to `F2t`, skipping the prologue. If the branch is taken again, execution stays in the F$.

Call emulation instructions are the result of translating procedure calls. Since a translation corresponding to the return application address may not exist or could be evicted before the translated program returns from the procedure, call emulation instructions explicitly set the return location as the original application return address. When the return happens, it is handled as an indirect branch. Call emulation instructions can be seen in fragment `F2`.

Link instructions transfer control to the translated target of a direct CTI. Trampoline instructions are overwritten to become link

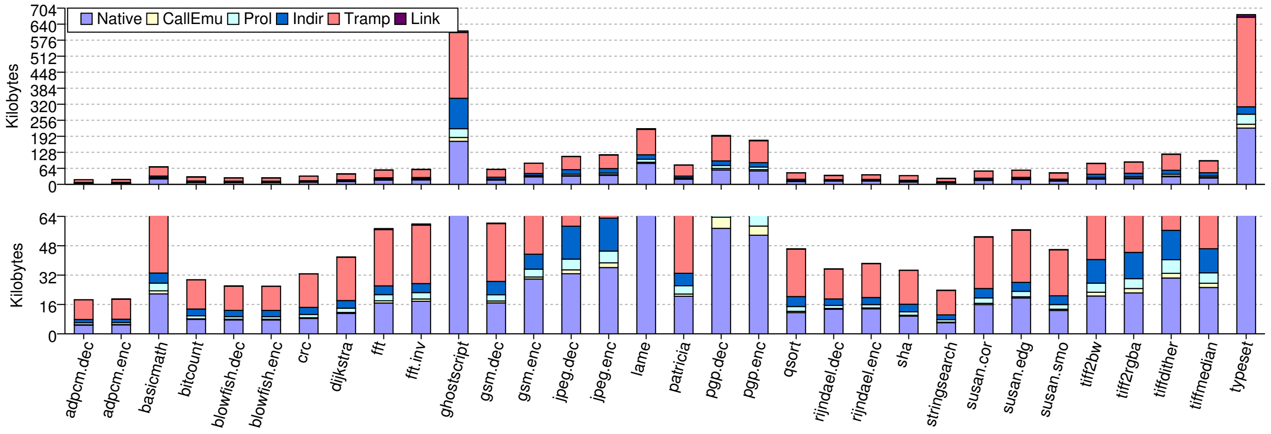


Figure 5: Translated code size for an unbounded F$ (different scales for clarity)

instructions when previously unseen application code is translated. The link instructions go to the location *after* the target fragment's prologue. A link instruction (`j F4t`) can be seen in fragment `F2` that transfers control to fragment `F4`, skipping the prologue.

Indirect handling instructions are emitted when an indirect CTI is translated. This code tries to map the application address in the target register to an existing F$ address. If the target application address is untranslated, the DBT is re-entered. Fragment `F4` ends with indirect handling code.

The native and call emulation instructions are the ones that advance program execution. The rest are "control code" introduced by the DBT to remain in control and ensure that untranslated code is processed prior to its execution.

4.2 Initial Generated Code Composition

We investigated how much of the code in the F$ corresponds to each instruction category. Figure 5 shows the amount of code generated per benchmark for each instruction category for an unbounded F$. When compared to Figure 3, the benchmarks without performance loss are those whose total amount of translated code fits in the bounded F$, such as *adpcm.enc/dec* and *stringsearch* for 64K and 32K.

The amount of code generated for some benchmarks greatly exceeds the capacity of the bounded F$ (e.g., *ghostscript* and *typeset*), even when considering only the native and call emulation categories. It is unlikely that such benchmarks ever run in a F$ size $\leq$ 64K without a considerable performance loss.

For several benchmarks, however, the native and call emulation instructions add to less than 64K (or even 32K and 16K). For instance, in *basicmath* and *patricia*, these two categories sum to less than 24K, with the rest of the code being control code. Nevertheless, these benchmarks suffer significant slowdowns with a bounded F$, as seen in Figure 3. Thus, they could benefit from a reduction in the amount of control code.

Figure 6 shows the relative utilization of the F$ by each instruction category for benchmarks that suffer at least a 50% slowdown for a 32K F$ with FLUSH (top) and FIFO (bottom). On average, native instructions account for less than 30% of the generated code: 27.42% with FLUSH and 26.07% with FIFO. Trampoline instructions are the largest consumer of F$ space, averaging 57.56% with FLUSH and 60.26% with FIFO (due to eviction preparation). Code used to handle indirect branches averages 7.27% with FLUSH and 6.86% with FIFO. Prologue code averages 5.56% for FLUSH and

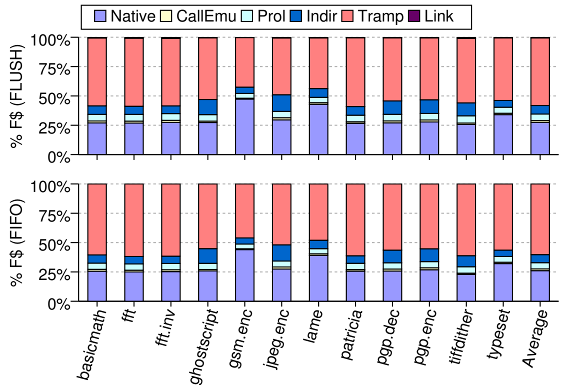


Figure 6: Initial relative F$-32K usage

5.28% for FIFO. Call emulation averages 1.58% for FLUSH and 1.53% for FIFO. Links average 0.6% with FLUSH and 0% with FIFO because we do not count links when they overwrite trampolines. For brevity, experimental results in the rest of this section are shown for the benchmarks in Figure 6 for a 32KB F$. In the final results, presented later, we show all the benchmarks with 16K, 32K and 64K F$ sizes with FLUSH and FIFO.

We now describe and evaluate our techniques to minimize the footprint of instructions in each "control code" category. The aim is to reduce the size of the generated code and improve the utilization of the F$ in favor of native instructions. Our experimental configuration forms DBBs, but larger code regions [11] can also be characterized in this manner. The total amount of code generated in such cases will be larger, due to duplication and speculation, but the percentage of "control code" should be similar and the miss ratio and performance improvements could be even better. We start reducing trampoline code, as it is the greatest consumer of F$ space.

4.3 Direct Branch Trampolines

The design of a trampoline is guided by the target architecture and the internal design of the DBT. Trampolines help perform a context switch to the DBT. The number of instructions required by a context switch depends on the target architecture (e.g., 22 on SPARC, 78-84 on MIPS and 10 on x86 [17]). To avoid unnece-

```
  .fcache
//builder(to_PC,frag)
Tr:sw  $a0,a0_ofs($sp)
   sw  $a1,a1_ofs($sp)
   lui  $a0,HI(to_PC)
   ori  $a0,$a0,LO(to_PC)
   lui  $a1,HI(&frag)
   ori  $a1,$a1,LO(&frag)
   j reenter
```

(a) 2-Argument

```
  .fcache
//builder(link)
Tr:sw  $a0,a0_ofs($sp)
   lui  $a0,HI(&link)
   ori  $a0,$a0,LO(&link)
   j reenter
```

```
  .datamem
//linker record
link: to_PC &frag
```

(b) 1-Argument

```
  .fcache
//builder(link)
Tr:sw  $ra,ra_ofs($sp)
   jal reenter
   &link
```

```
  .translator
reenter:
   sw $a0_ofs($sp)
   lw $a0,0($ra)
   ...
```

(c) Contiguous Data

```
  .fcache
//builder(Tr)
Tr:sw  $ra,ra_ofs($sp)
   jal reenter
```

```
  .datamem
//trampoline map
tramp:Tr &link
```

(d) Mapped Data

```
  .fcache
//after $ra def.
   lui  $t0,HI(&shra)
   ori  $t0,$t0,LO(&shra)
   sw  $ra,0($t0)
   ...
//builder(Tr)
Tr:jal reenter
```

```
  .translator
reenter:
   ...
   lui  $t0,HI(&shra)
   ori  $t0,$t0,LO(&shra)
   lw  $ra,0($t0)
   sw  $ra,ra_ofs($sp)
   ...
```

(e) Shadow Link Register

Figure 7: Trampoline design choices

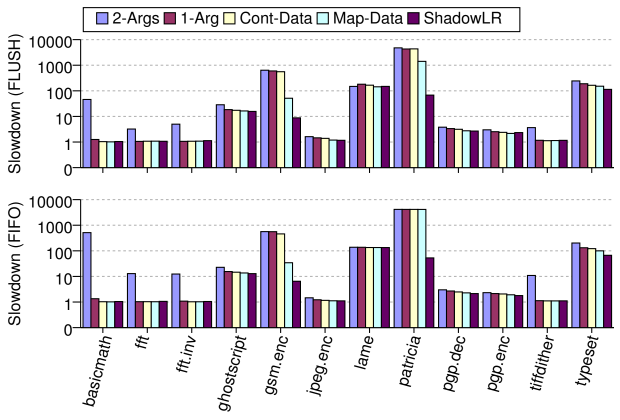


Figure 8: Performance of trampoline designs

sary F$ pressure, most context save instructions on Strata/PISA are factored into a single "re-entrance routine" (the entry point to the fragment builder). Each trampoline needs only to perform a partial context save before jumping to the re-entrance routine. This approach is natural for a small, bounded F$. However, there are other unique opportunities to reduce the size of the trampoline. We consider the alternative designs shown in Figure 7. They are ordered by trampoline size. A code reduction is achieved by moving the information associated with the trampoline into data memory to free F$ space, which increases the execution cost of the trampoline.

Design (a), "2-argument", is the one used in Strata by default. The trampoline conveys two pieces of information to the builder: the application address to translate and a pointer to the fragment map entry associated with the fragment invoking the DBT. Both arguments depend on the trampoline and are set by it. A partial context save is needed to free the argument registers before setting the values and jumping to the re-entrance routine. With this approach the builder is accessed with the necessary arguments directly after the context save, trading F$ space for a smaller dynamic instruction count.

Design (b), "1-argument", exploits the fact that the fragment linker also records the target address and source fragment of each trampoline. It passes to the builder only a pointer to the appropriate link record (`&link`). This approach uses less instructions in the F$ but needs an extra step to enter the translator; i.e., to retrieve the trampoline information from data memory.

On many architectures, loading a constant pointer takes more than one instruction. Instead, design (c), "contiguous data", stores the link record pointer (as data) in the instruction slot immediately after the trampoline. A jump-and-link (`jal`) instruction is then used to access the re-entrance routine, which sets the argument using a load relative to the value in the link register (`$ra`). The tradeoff is saving one instruction but polluting the data cache with trampoline data mixed with code.

This pollution can be avoided by storing the trampoline data in main memory, which saves more F$ space. For this purpose, a hash table indexed by trampoline address is used to store and recover the trampoline data. Design (d), "mapped data", implements this approach. It trades data memory (an extra hash table) for F$ space, and requires a hash lookup on re-entering the builder.

Instead of spilling the link register in every trampoline before overwriting its value (with the jump-and-link), Design (e), "shadow link register" (SLR), can be used. This approach requires the DBT to identify the instructions that change the value of `$ra` and insert code to update the value of a shadow variable (`shra`). Then, trampolines can safely overwrite `$ra` since the re-entrance routine uses the value in `shra` to perform the context save. The tradeoff is that if the application code changes the value of `$ra` too often, the translated code size could be increased. Fortunately, `$ra` is typically defined only when a non-leaf procedure passes the return address to a callee and when it recovers its own return address from the stack. Because the number of calls and returns from non-leaf procedures is usually smaller than the number of CTIs in need of a trampoline, this technique can be very effective.

Evaluation

Figure 8 shows the performance of the benchmarks for a 32K F$ relative to an unbounded F$. For some benchmarks, the initial gain obtained with "1-Argument" is significant: *basicmath* goes from 46x slowdown with FLUSH and 514x slowdown with FIFO to just 1.26x and 1.35x! Once close to the ideal, the improvements are less impressive: *fft* goes from initial slowdowns of 3.2x and 12.9x with FLUSH and FIFO to 1.06x and 1.03x for "1-Argument". The other designs do not achieve further improvement. In benchmarks with high F$ pressure, the effect is progressive: *ghostscript* has slowdowns of 28.6x (2-Arg), 18.5x (1-Arg), 17.4x (Cont.Data), 16.4x (Map.Data) and 15.7x (ShadowLR) with FLUSH. *patricia* has slowdowns of 68x with FLUSH and 53x with FIFO when using SLR. However, the other designs have slowdowns beyond 1000x. The greatest improvement overall is achieved with SLR.

Figure 9 shows the 32K F$ utilization after applying "Shadow Link Register". With both FLUSH and FIFO, native instructions now account for 58.8% of the F$ on average, while trampoline

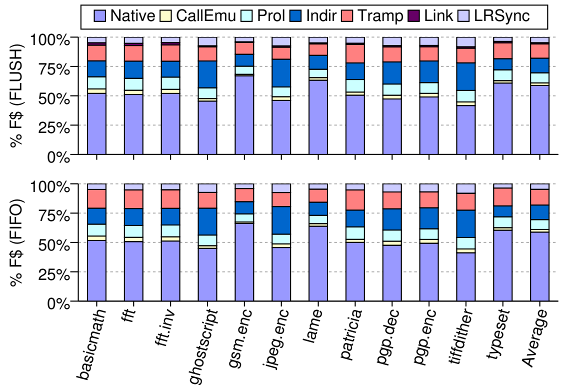


Figure 9: Relative F$-32K usage after Shadow LR

instructions are reduced to an average of 12.18% (FLUSH) and 13.42% (FIFO). We introduce a new category ("LR Sync") to account for the code introduced to update the shadow variable; it averages 4.74% with FLUSH and 4.68% with FIFO. From these results we conclude that when F$ pressure is high, even a small reduction in trampoline size has a dramatic effect on performance due to improved F$-32K usage. Since "Shadow Link Register" (SLR) has the best overall performance, we use it in the rest of the paper.

4.4 Indirect Branch Handling

After reducing trampoline size, indirect branches become a more important source of F$ pressure. In Figure 9, the amount of code generated for indirect branch handling is about the same as trampoline code: 12.4% on average for both FLUSH and FIFO. We now show how to minimize the indirect branch handling code.

An indirect CTI (branch, call or return) may have multiple runtime targets, so it can not be directly linked. On the other hand, doing a context switch to let the DBT find the translated target every time the indirect is executed degrades performance. The context switch should ideally occur only if the application target address does not have a corresponding translation in the F$. Several mechanisms have been proposed to map the original application address to a translated address without leaving the F$, saving the cost of a full context-switch. Past work has shown that the most useful technique across platforms is the *Indirect Branch Table Cache* (IBTC) [12]. The IBTC is a small, direct-mapped table that associates indirect branch target addresses to their F$ locations. It is allocated in main memory and does not consume F$ space.

For every indirect CTI, code is generated in the F$ to access the IBTC, as shown in Figure 10(a). The code first spills registers to safely do hash table computations. The table lookup is done next. If a match is found (a "hit"), the IBTC holds a corresponding F$ address. On a hit, the registers, except `$ra`, are restored. `$ra` is used to jump to the target fragment and is restored by that fragment's prologue. If no match is found (a "miss"), the DBT is entered.

Emitting an IBTC lookup in every fragment (ending with an indirect) puts extra pressure on the F$. Our approach trades dynamic instruction count for more compact code with a single "Out-Of-Line IBTC Lookup" (OOL-Lookup) as shown in Figure 10(b). The shared out-of-line lookup code is similar to a function call – arguments are passed to the code to indicate the requested application address and a pointer to the fragment map's record of the fragment with the indirect branch (to pass to the builder on a miss).

An equivalent of the "contiguous data" trampoline for indirects

```
     .fcache
//for each indirect
     sw $a0,a0_ofs($sp)
     sw $a1,a1_ofs($sp)
     sw $ra,ra_ofs($sp)
     add $a0,$z0,$rt
lkup: //$ra = &table
     //$a1 = hash($a0)
     //$ra = $ra[$a1]
     lw $a1,PC_ofs($ra)
     bne $a1,$a0,miss
 hit: lw $ra,FPC_ofs($ra)
     lw $a0,a0_ofs($sp)
     lw $a1,a1_ofs($sp)
     jr $ra
miss: lui $a1,HI(&frag)
     ori $a1,$a1,LO(&frag)
     j reenter_ibtc
```

(a)Inline IBTC Lookup

```
     .fcache
//for each indirect
     sw $a0,a0_ofs($sp)
     sw $a1,a1_ofs($sp)
     add $a0,$z0,$rt
     lui $a1,HI(&frag)
     ori $a1,$a1,LO(&frag)
     j lkup

//shared by all indirects
lkup: sw $ra,ra_ofs($sp)
     sw $a1,at_ofs($sp)
     //$ra = &table
     //$a1 = hash($a0)
     ...
miss: lw $a1,at_ofs($sp)
     j reenter_ibtc
```

(b) Out-of-line IBTC Lookup

```
     .fcache
//for each indirect
     sw $ra,ra_ofs($sp)
     sw $a0,a0_ofs($sp)
     add $a0,$z0,$rt
     jal lkup
     &frag

//shared by all indirects
lkup: sw $a1,a1_ofs($sp)
     lw $a1,0($ra)
     sw $a1,at_ofs($sp)
     //$ra = &table
     //$a1 = hash($a0)
     ...
miss: lw $a1,at_ofs($sp)
     j  reenter_ibtc
```

(c) Contiguous Data Indirect

```
     .fcache
//for each indirect
     sw $ra,ra_ofs($sp)
     jal l$rt
     &frag

//shared by $rt-indirects
l$rt: sw $a0,a0_ofs($sp)
     add $a0,$z0,$rt
     j lkup
//shared by returns
l$ra: sw $a0,a0_ofs($sp)
     lw $a0,ra_ofs($sp)
//shared by all indirects
lkup: sw $a1,a1_ofs($sp)
     lw $a1,0($ra)
     ...
```

(d) Shared Target Register Copies

Figure 10: Indirect branch handling with an IBTC

can also be implemented, as shown in Figure 10(c). The "contiguous data indirect" uses a jump-and-link (`jal`) instruction to access the out-of-line IBTC lookup. The address of the requesting fragment's record is put in the instruction slot after the `jal`. In the lookup code, the value in the link register (`$ra`) is used to load that pointer into the appropriate register.

The out-of-line IBTC lookup code uses a fixed argument register (`$a0`) for the target application address. To do a lookup, the register (`$rt`) that contains the address must be copied to the argument register. For further gain, our approach, called "Shared Target Register Copies" (STRC), shares the code that performs this copy among all indirects that use the same register. As shown in Figure 10(d), the code generated for each indirect spills `$ra` and uses a `jal` to go to an entry point in the out-of-line shared code that depends on the target register (`$rt`) used. For each unique `$rt`, a single transfer routine spills the argument register (`$a0`), copies `$rt` to `$a0` and jumps to the IBTC lookup code. The transfer routines are emitted on-demand as new indirect target registers are discovered by the DBT.

Evaluation

Figure 11 shows the performance of the benchmarks with the different IBTC lookups. *patricia* has the greatest improvement: with FLUSH, the slowdown of 68x after SLR is reduced to 1.87x (OOL-Lookup), 1.31x (cont.data) and 1.23x (STRC). *ghostscript* shows more steady slowdown reductions: with FLUSH, from 15.7x after SLR to 14.37x (OOL-Lookup), 14.16x (cont.data) and 13.47

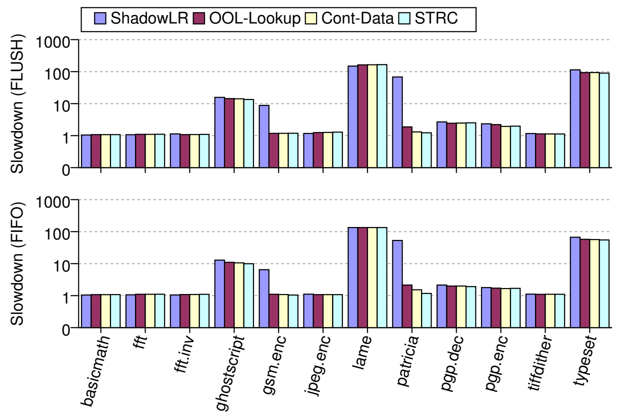


Figure 11: Performance of IBTC lookups

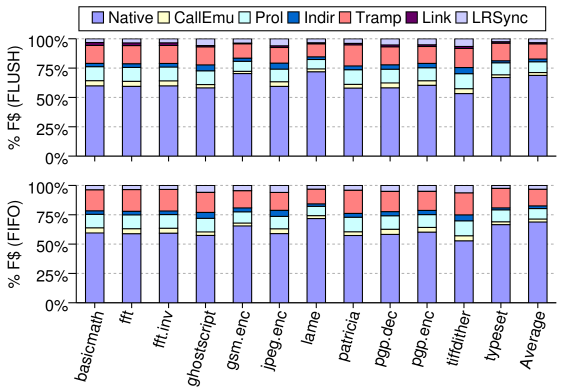
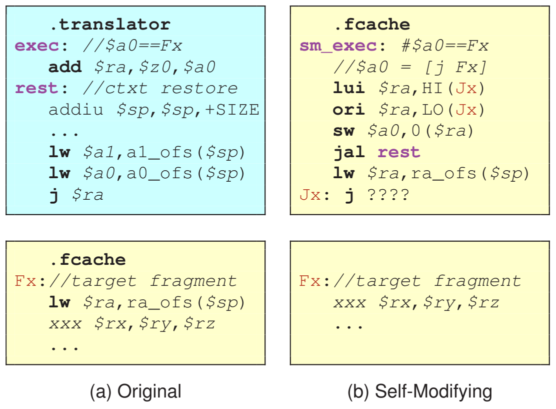


Figure 12: Relative F$-32K usage after STRC



```
    .translator
exec: //$a0==Fx
    add $ra,$z0,$a0
rest: //ctxt restore
    addiu $sp,$sp,+SIZE
    ...
    lw $a1,a1_ofs($sp)
    lw $a0,a0_ofs($sp)
    j $ra
```

```
    .fcache
sm_exec: #$a0==Fx
    //$a0 = [j Fx]
    lui $ra,HI(Jx)
    ori $ra,LO(Jx)
    sw $a0,0($ra)
    jal rest
    lw $ra,ra_ofs($sp)
Jx: j ????
```

```
    .fcache
Fx://target fragment
    lw $ra,ra_ofs($sp)
    xxx $rx,$ry,$rz
    ...
```

```
Fx://target fragment
    xxx $rx,$ry,$rz
    ...
```

Figure 13: Control transfer to fragment

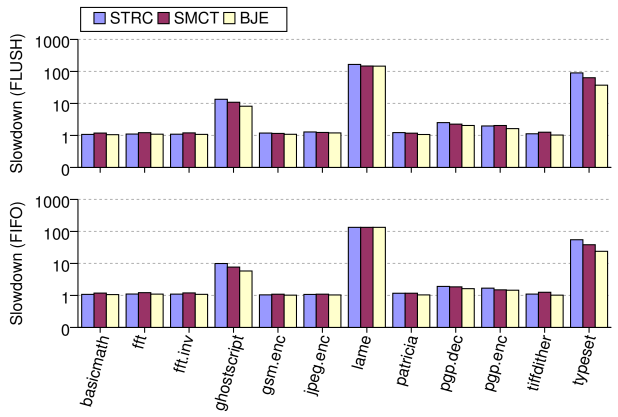


Figure 14: Performance with SMCS and BJE

(STRC). FIFO has similar trends. The benchmarks that already fit, however, suffer some degradation due to the increased number of jumps: `basicmath` goes from 5% overhead after SLR to 8% (OOL-Lookup), 7% (cont.data) and 8% (STRC) with both FLUSH and FIFO.

Figure 12 shows the 32K F$ utilization after applying "Shared TR Copies". With both FLUSH and FIFO, native instructions now account for 68.8% of the F$ (an increase of 10% after using only SLR), while indirect handling code is reduced to an average of 2.35% with FLUSH and 2.27% with FIFO. In the rest of the paper, we use "Shared Target Register Copies" (STRC) because it achieves the greatest improvement.

4.5 Prologue Elimination

After reducing the indirect branch handling code, fragment prologue instructions now account for an average 9% of the F$. In Strata/PISA, the fragment prologue restores the link register (`$ra`), used to transfer control to the fragment from the DBT with an indirect jump (as shown in Figure 13(a)). On an IBTC hit, restoring that register is also left to the fragment.

The indirect jump can be eliminated by rewriting a direct jump. This approach, which we call "Self-Modifying Control Transfer" (SMCT), is illustrated in Figure 13(b). As shown, instead of ending with an indirect jump, the routine that returns control to the F$ rewrites its last instruction to be a direct jump to the target fragment. In systems with instruction and data caches, self-modifying code requires synchronization between the caches. Depending on architectural details, this operation can be expensive since it may require flushing a cache line or the *entire* instruction cache. However, in many embedded designs with SPM, the SPM addresses are not cached and a data write is immediately sent to the SPM. As a result, synchronization is not needed and self-modifying code is inexpensive. Because Strata is in ROM, our implementation initially emits a "return routine" into the F$ on start-up. This routine transfers control from the DBT to the F$, by rewriting the jump at the end as described above. The code for an IBTC hit must also be modified to work with self-modifying code: to go to the target fragment, a direct jump is overwritten with the target fragment address found in the IBTC.

After eliminating the prologue, it does not have to be skipped when fragments are linked. Now, when the last instruction in a fragment transfers control to the fragment immediately after it, we can elide that jump. "Bottom Jump Eliding" (BJE) implements this idea.

Evaluation

Figure 14 shows the performance of the benchmarks with both SMCT and BJE. High-pressure benchmarks have significant improvements: *ghostscript* for FLUSH goes from a 13.47x slowdown (STRC) to 10.88x with SMCT and 8.18x with BJE. With FIFO it goes from 9.93x (STRC) to 7.68x (SMCT) and 5.8x (BJE). In some

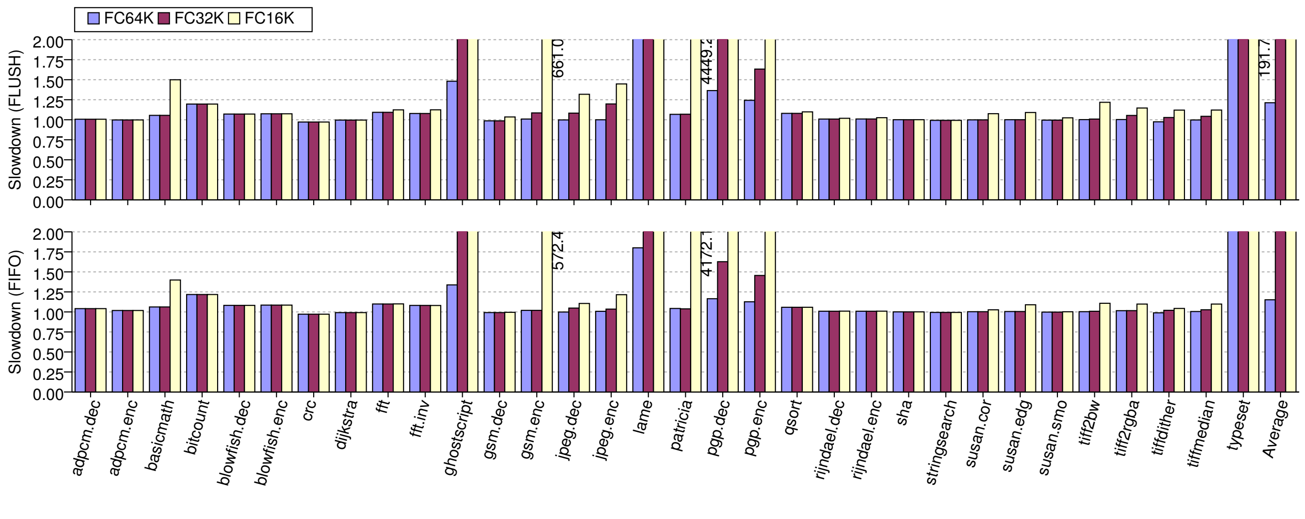


Figure 16: Final performance normalized to unbounded F$

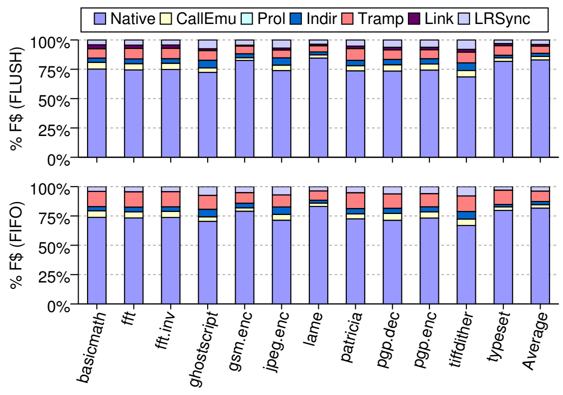


Figure 15: Relative F$-32K usage after BJE

benchmarks, SMCT increases the slowdown and BJE reduces it again. This is the case with *pgp.encode* with FLUSH. It has a 1.98x slowdown after STRC, which increases to 2.04x after SMCT. With BJE, it goes to 1.63x.

Figure 15 shows the F$ distribution after BJE. In this case, the replaced links are discounted and the prologue has been eliminated, leaving more room for native instructions. They now average 83% (FLUSH) and 81.7% (FIFO). For the final results, we use the "self-modifying control transfer" and "bottom jump eliding", since this combination has the better performance impact.

5. OVERALL RESULT

Figure 16 shows the slowdowns of all benchmarks after applying the techniques in this paper. The results are normalized to the initial unbounded F$. Our techniques achieve significant improvements when the *translated* code working set did not fit initially in the F$ due to excessive DBT control code. For example, *basicmath* initially had a 4983.96x slowdown for a 16K F$ and 46x in a 32K F$ with FLUSH. After applying our techniques, the overhead for both sizes was reduced to only 5%. This final overhead is the same for a 64K F$. However, it initially was only 1%, indicating a performance cost associated with our techniques that is not amortized when the code fits in the F$. *patricia* has an impressive improvement. For a 32K F$ with FLUSH: its initial slowdown of 4769.5x is reduced to 1.07x. With FIFO, the initial slowdown in 32K F$ was 4159x, but our techniques reduce it to 1.04x. The performance is nearly equivalent (within 7% for FLUSH and 4% for FIFO) to the ideal case of an unbounded F$. *dijkstra* is another example where our techniques make the final performance equivalent to the unbounded F$, after an initial 20.33x slowdown.

Although our techniques improve performance, there are situations where the slowdowns can not be overcome. In a 16K F$ with FIFO, the initial slowdown of 4416.92x for *patricia* is reduced to only 4172.06x. Our techniques help somewhat, but can not fully overcome the performance degradation when the *application* working code set does not fit.

Our techniques reduce code size and lead to a performance improvement regardless of the F$ management technique. The initial average slowdowns for FLUSH of 10x (64KB) is reduced to 1.21x, the slowdown of 184.88x (32KB) is reduced to 6.99x and the slowdown of 643.28x (16KB) is reduced to 171.98x (16KB). For FIFO, the average slowdowns of 9.35x (64KB), 176.96x (32KB), and 433.72x (16KB) are reduced to 1.15x (64KB), 6.09x (32KB) and 158.51x (16KB).

6. RELATED WORK

Several studies address the management of bounded software code caches. For general-purpose systems, Bala et al. propose preemptively flushing the trace cache of a dynamic optimizer when detecting program phase changes [3]. Bruening and Amarasinghe develop methods for maintaining the consistency of the code cache in the presence of self-modifying code and for dynamically increasing the size of the code cache as the working set grows [4]. Hazelwood and Smith explore different eviction policies for the trace cache of a dynamic optimizer and conclude that FIFO reduces the miss rate over FLUSH by half [9]. Later, they find that medium-grained evictions scale better than FLUSH and FIFO [10]. From observing the lifetime of traces, they also develop a generational scheme that stores short-lived and long-lived traces in separate caches. Guha et al. adapt generational cache management to embedded systems, in order to reduce the maximum dynamic size of an *unbounded* code cache [7]. Baiocchi et al. manage constrained code caches by keeping a compressed F$ region to which fragments are initially evicted and decompressed if required again [2]. This technique reduces the retranslation cost for accessing Flash memory. They also "pin" decompressed fragments to avoid repeated compressions and decompressions.

Reducing the size of DBT's "control code" has been studied by Guha et al., who focus *only* on trampolines [6]. In their setup, fragments and trampolines are inserted from opposite ends of the code cache. They reduce the trampoline size and delete free trampolines on top of the stack. They also unify trampolines that request the same address. Their optimizations reduce the relative size of trampolines from 66.7% to 41.4%, for an *unbounded* code cache. Our techniques consider *all* control code and do considerably better when applied together. Hiser et al. study the allocation of trampolines in a separate region (a "trampoline pool") in comparison to interleaving them with fragments [11]. They find the technique lowers I-cache pressure (over 18% of the misses are eliminated with a pool). The reduction of trampoline size to 1 instruction, thanks to the "Shadow Link Register", combined with "Bottom Jump Eliding", fully eliminates the need for trampoline management.

A similar concept to the "Shadow Link Register" technique is used by Miller and Agarwal in Flexicache [15]. It is a system that provides a software-managed code cache for processors with SPM and no I-cache. Flexicache is not a DBT; it uses a binary rewriter to *statically* form cache blocks of *fixed* size and inject control code *prior* to the program's execution. The rewriter also inserts code to update the shadow variable after each definition of the link register (including call sites) and to load the value of the shadow variable before each use. Instead, we move the update to the beginning of the callee and do not instrument uses, because the proper value of the link register is restored on return from the DBT. Our work shows that a DBT with a *bounded* F$ allocated to SPM can provide similar functionality as Flexicache, while enabling other services.

Hiser et al. study several indirect branch handling techniques across platforms [12]. They found IBTC to be the most useful technique, but that the placement of the lookup code has little importance in general-purpose systems. Our work shows that in embedded systems aggressive factorization of the lookup code can benefit performance due to space savings.

7. CONCLUSION

This paper investigated the composition of the code generated by a DBT and proposed techniques to reduce its footprint, targeting "control code". We described and evaluated methods to minimize the space needed by trampolines, indirect branch handling and context switch code. With these techniques, the translated program code fits better in a small, bounded code cache for an embedded system. In general, when our approaches are enabled, a DBT with a constrained code cache has similar performance than a DBT with an unbounded code cache.

8. ACKNOWLEDGMENTS

This research was supported by the National Science Foundation under grants CNS-0720483, CCF-0702236, CNS-0551492, CNS-0720789 and CNS-0716478.

9. REFERENCES

[1] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35:59–67, 2002.

[2] J. Baiocchi, B. R. Childers, J. W. Davidson, J. D. Hiser, and J. Misurda. Fragment cache management for dynamic binary translators in embedded systems with scratchpad. In *Int'l. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2007.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Conf. on Programming Language Design and Implementation (PLDI)*, 2000.

[4] D. Bruening and S. Amarasinghe. Maintaining consistency and bounding capacity of software code caches. In *Int'l. Symp. on Code Generation and Optimization (CGO)*, 2005.

[5] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. Deli: a new run-time control point. In *Int'l. Symp. on Microarchitecture (MICRO)*, 2002.

[6] A. Guha, K. Hazelwood, and M. L. Soffa. Reducing exit stub memory consumption in code caches. In *Int'l. Conf. on High Performance Embedded Architectures and Compilers (HiPEAC)*, 2007.

[7] A. Guha, K. Hazelwood, and M. L. Soffa. Code lifetime-based memory reduction for virtual execution environments. In *Workshop on Optimizations for DSP and Embedded Systems*, 2008.

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE Workshop on Workload Characterization*, 2001.

[9] K. Hazelwood and M. D. Smith. Code cache management schemes for dynamic optimizers. In *Workshop on Interaction between Compilers and Computer Architectures*, 2002.

[10] K. Hazelwood and M. D. Smith. Managing bounded code caches in dynamic binary optimization systems. *ACM Trans. Archit. Code Optim.*, 3:263–294, 2006.

[11] J. D. Hiser, D. Williams, A. Filipi, J. W. Davidson, and B. R. Childers. Evaluating fragment construction policies for SDT systems. In *Int'l. Conf. on Virtual Execution Environments (VEE)*, 2006.

[12] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *Int'l. Symp. on Code Generation and Optimization (CGO)*, 2007.

[13] Intel Corporation. *Intel PXA27x Processor Family Developer's Manual*, 2006.

[14] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symp.*, 2002.

[15] J. E. Miller and A. Agarwal. Software-based instruction caching for embedded processors. In *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[16] K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *Annual Computer Security Applications Conf.*, 2002.

[17] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Int'l. Symp. on Code Generation and Optimization (CGO)*, 2003.

[18] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *Int'l. Symp. on Microarchitecture (MICRO)*, 2005.