Enabling Dynamic Binary Translation in Embedded Systems with Scratchpad Memory

José A. Baiocchi, University of Pittsburgh
Bruce R. Childers, University of Pittsburgh
Jack W. Davidson, University of Virginia
Jason D. Hiser, University of Virginia

Important challenges for embedded systems can be addressed by dynamic binary translation. A dynamic binary translator stores translated instructions in a software-managed code cache, which is usually large to minimize overhead. This article shows how to use a small scratchpad memory for the code cache. A small code cache may require frequent code evictions and retranslation, which degrade performance. We propose techniques to reduce the number of instructions inserted by the translator and a way to form fragments that minimizes translated code size. With our techniques, a much smaller code cache can hold a program's translated code working set.

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: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Code Generation, Dynamic Binary Translation, Footprint Reduction, System-on-Chip

ACM Reference Format:

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 [Desoli et al. 2002], security [Kiriansky et al. 2002; Hu et al. 2006], power management [Wu et al. 2005], software instruction caching [Miller and Agarwal 2006; Baiocchi et al. 2007], code decompression [Shogan and Childers 2004] and Flash demand paging [Baiocchi and Childers 2011]. For instance, a DBT can improve performance in embedded systems with Flash memory by...
incrementally loading the program binary image, which requires fewer accesses to the slow Flash device, rather than loading the whole binary into main memory (DRAM) all at once. Previous work with a DBT for this type of system has achieved a 1.9x to 2.2x speedup over native execution [Baiocchi et al. 2007]. Despite these promising uses, the adoption of DBT in 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 [Hiser et al. 2011].

Unfortunately, a large, unbounded code cache can easily exceed the limited capacity of an embedded system’s memory resources. The memory resources may include a small on-chip scratchpad memory (SPM), which could be profitably employed to hold the code cache due to the SPM’s single cycle access and low power [Banakar et al. 2002]. However, the small SPM is directly at odds with the need for a large code cache. As a result, the performance of a dynamically translated program can be poor. The problem is that limiting the size of the code cache to the SPM size makes it unlikely that the translated code working set will fit. When it is necessary to obtain space for newly translated code and the code cache is full, some of the existing translated code must be evicted. It is possible that the evicted code is needed again, and capacity misses appear.

DBTs for general-purpose systems employ specific region formation strategies [Hiser et al. 2006] designed to improve locality and reduce the dynamic instruction count. To that end, these strategies often duplicate translated code and speculatively translate code that is not guaranteed to be executed in the near future. Duplication and speculation increase pressure on a bounded code cache, leading to an increased frequency of capacity overflows and a higher code cache miss rate. We show carefully selecting region formation policies that minimize code duplication and speculation helps prevent unnecessary translated code growth.

Past research for general-purpose systems examined eviction schemes that effectively decide what code to keep in the code cache and when to discard it [Bala et al. 2000; Hazelwood and Smith 2002; 2006]. Even with these techniques, the code cache size can still be hundreds of kilobytes to a few megabytes [Hazelwood and Smith 2006]. For embedded systems, compression and pinning can be used to keep needed code in the code cache [Baiocchi et al. 2007]. While these approaches are important, they do not tackle the footprint of the translated code. Some approaches do tackle the footprint of the translated code. Guha et al. [2007; 2008] look at reducing the size and number of exit stubs that are necessary for the DBT to remain in control, and removing unneeded fragments from the code cache. We systematically show in this article the breakdown of code inserted by the DBT for its own purposes and find that it 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. A smaller translated code footprint reduces pressure on the code cache and improves its miss ratio. We categorize translated instructions 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 instruction category and identify which aspects of the DBT’s “control code” have the largest impact on 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. We show this can significantly improve performance.

This work makes the following contributions:

— A classification of the instructions generated by a DBT, used to characterize the utilization of the code cache. This study has not previously been done in the context of embedded systems.

— 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 with a small scratchpad.

— Descriptions and implementations of novel techniques for substantially reducing the space consumed by DBT-injected code, including single-instruction exit stubs, factored indirect handling, fragment prologue elimination, and bottom jump eliding.

— A study of exit stub pooling with different pool management strategies, and its comparison to single-instruction exit stubs.

— Experimental evidence of the negative impact of code duplication and speculative translation in a bounded code cache; as well as the selection of a region formation policy that minimizes the impact.

— Comprehensive experimental evaluation of the techniques, considering different code cache sizes and eviction policies.

The rest of the article 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 5 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 6 studies the impact of different fragment formation policies on code footprint. Section 7 gives the overall improvement with the selected techniques. Section 9 describes related work and Section 10 concludes.

2. DBT ON A SYSTEM-ON-A-CHIP

We start with a description of the type of embedded system targeted by our study and how a DBT is integrated with it.

2.1. Target Embedded System

Our techniques target a System-on-a-Chip (SoC) similar to the example shown in Figure 1. An embedded SoC may be custom for a particular application, or more likely, an application domain (i.e., a platform-based design). The SoC in the figure has a processor, L1 data cache (D-cache), an application-specific integrated circuit (ASIC), scratchpad memory (on-chip SRAM), ROM (on-chip NOR Flash), controllers for external SDRAM and Flash memories and off-chip I/O channels. There is an on-chip communications fabric interconnecting the components. The external SDRAM and Flash memories are also shown.

The SDRAM is used as main memory and holds application data during program execution. The Flash memory is managed as a file system by the operating system (OS) and holds user files, including application binaries. Code in Flash must be loaded into the SPM to be executed. A hardware L1 instruction cache is unnecessary because code executes directly from the SPM.

In the SoC just described, a Dynamic Binary Translator (DBT) is kept and executed in ROM as part of the system code. The DBT’s data structures are placed in main
memory and share the L1 data cache with the application. The DBT’s code cache is allocated to the SPM [Baiocchi et al. 2007] to benefit from its low latency (1 cycle). Thus, the DBT provides a form of software instruction caching [Miller and Agarwal 2006]. We assume that applications are not aware of the SPM at compile-time, i.e., the program has not been compiled to use the SPM for code or data. Such an approach [Verma and Marwedel 2006; Udayakumaran et al. 2006] could be supported by assigning part of the SPM to the compiled code and a part to the DBT. A DBT can also provide other services like on-demand code decompression [Shogan and Childers 2004], power management [Wu et al. 2005] and secure execution [Kiriansky et al. 2002]. We now describe the operation of a DBT.

2.2. DBT operation

Figure 2 shows a high-level view of a typical DBT. The DBT mediates application execution, i.e., all application instructions are examined, and possibly modified, prior to their first execution. Thus, 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. 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 fragment’s application address as a key. During translation, the code in the fragment is written to the code cache, known as a fragment cache (F$). When the DBT finishes translation, the application context is restored and control is transferred to the new fragment.

The Fragment Builder is the DBT’s component that fetches, classifies and translates instructions until a stop condition is met [Hiser et al. 2006]. 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 end at all CTIs to form Dynamic Basic Blocks (DBB) [Smith and Nair 2005].

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, fragment linking is used. When a new fragment is created, all trampolines that request its application address are “patched” to jump directly to the fragment [Bala et al. 2000]. For this purpose, the DBT’s Fragment Linker records all trampolines and their target application addresses. 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 (IBTC), is employed to map target application addresses to their corresponding translated addresses [Hiser et al. 2011].

When a F$ is too small and cannot hold the full translated code working set of the program, it eventually overflows. F$ management techniques are employed to handle this problem [Bala et al. 2000; Hazelwood and Smith 2002; 2006; Baiocchi et al. 2007]. These solutions preemptively or on-demand discard some or all fragments to make room for new code in the F$. However, they are likely to increase the F$ miss rate because discarded code may be requested again. The performance penalty due to re-translation can be high, especially in embedded systems where the binary image resides on slow Flash memory [Baiocchi et al. 2007].

In this article, we investigate 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.

2.3. Experimental Methodology

For this investigation, we used Strata [Scott et al. 2003], a publicly available DBT. Strata was retargeted to SimpleScalar/PISA [Austin et al. 2002] and modified to include the techniques in this article. We run programs from MiBench [Guthaus et al. 2001] that SimpleScalar can execute, with their large input data sets.

We extended SimpleScalar’s cycle-accurate simulator with Flash and SPM. We use the configuration shown in Table I, which models the Intel/Marvell 624MHz XScale PXA-270 processor [Intel Corp. 2006], augmented with SPM, SDRAM and NOR Flash. 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. The simulator allows us to model different SPM configurations.

---

1 We could not run mad, rsynth, and sphinx due to incompatibilities with SimpleScalar.
Table I. SimpleScalar Configuration

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>XScale PXA-270 @ 624 Mhz</td>
<td></td>
</tr>
<tr>
<td>fetch:ifqsize</td>
<td>8</td>
</tr>
<tr>
<td>fetch:xplat</td>
<td>3</td>
</tr>
<tr>
<td>fetch:sm</td>
<td>1</td>
</tr>
<tr>
<td>fetch:speed</td>
<td>1</td>
</tr>
<tr>
<td>bpred:bimod</td>
<td>128</td>
</tr>
<tr>
<td>bpred:bimod</td>
<td>128</td>
</tr>
<tr>
<td>bpred:btb</td>
<td>512 4</td>
</tr>
<tr>
<td>decode:wif</td>
<td>1</td>
</tr>
<tr>
<td>issue:inorder</td>
<td>true</td>
</tr>
<tr>
<td>issue:wrongpath</td>
<td>true</td>
</tr>
<tr>
<td>tlb:dtlb</td>
<td>1:4096:32:f</td>
</tr>
<tr>
<td>tlb:itlb</td>
<td>1:8192:32:f</td>
</tr>
<tr>
<td>cache:il1</td>
<td>none</td>
</tr>
<tr>
<td>cache:il1</td>
<td>32:32:32:f</td>
</tr>
<tr>
<td>cache:dl1</td>
<td>none</td>
</tr>
<tr>
<td>cache:dl1</td>
<td>none</td>
</tr>
<tr>
<td>cache:dl1</td>
<td>none</td>
</tr>
<tr>
<td>cache:dl1</td>
<td>none</td>
</tr>
<tr>
<td>memport</td>
<td>1</td>
</tr>
<tr>
<td>res:ialu</td>
<td>1</td>
</tr>
<tr>
<td>res:imult</td>
<td>1</td>
</tr>
<tr>
<td>res:fpalu</td>
<td>1</td>
</tr>
<tr>
<td>res:fpmult</td>
<td>1</td>
</tr>
<tr>
<td>lsq:size</td>
<td>4</td>
</tr>
<tr>
<td>memport</td>
<td>4</td>
</tr>
<tr>
<td>commit:width</td>
<td>2</td>
</tr>
<tr>
<td>scratchpad:lat</td>
<td>1</td>
</tr>
<tr>
<td>tlb:lat</td>
<td>30</td>
</tr>
<tr>
<td>cache:dl1</td>
<td>60 12</td>
</tr>
<tr>
<td>flash:lat</td>
<td>900K 42K</td>
</tr>
</tbody>
</table>

SimpleScalar’s instruction set, called PISA, is similar to MIPS but to facilitate experimentation it uses a 64-bit instruction word. It includes a 16-bit annotation field and 8-bit register fields. For instance, we take advantage of the 16-bit annotation field to flag executed instructions in the F$ and measure the amount of translated code never executed. In reality, embedded processors use 16-bit or 32-bit instructions, so we double the SPM size for the simulations but refer to the smaller effective size (e.g., a 32K SPM is simulated with 64K).

To avoid expensive context switches, Strata uses fragment linking and provides several techniques for handling indirects. In our experiments, we use a shared IBTC, i.e., a single table in main memory accessed by all indirect CTIs. This technique has been determined to be the most useful across platforms [Hiser et al. 2011; Moore et al. 2009].

PISA returns (jr $ra) are treated as indirect branches.

### 3. IMPACT OF MEMORY CONSTRAINT

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

#### 3.1. Bounding the Size of the Fragment Cache

The lowest DBT overhead is achieved when the F$ is allowed to grow as needed, so code has to be translated only once, i.e., the F$ only suffers from compulsory misses. The size of the translated code when the F$ is unbounded is called the natural size of the F$. Bounding the F$ size to a fraction of its natural size reduces the overall memory overhead of DBT [Bruening 2004]. This is due not only to the limited size of the F$, but the size of the associated data structures is correlated to the number of fragments and trampolines the DBT must track.

When the F$ size is bounded, F$ overflows may occur. To handle them, we employ two techniques in our experiments, FLUSH and FIFO. These two techniques are at opposite ends of the spectrum of eviction granularities and performance cost [Hazelwood and Smith 2006]. 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 discarding the entire contents of the F$. The DBT must translate again any discarded fragment if necessary [Bala et al. 2000]. FIFO is a fine-grained technique that removes only the least recently created fragment(s) from the F$. FIFO treats the F$ as a circular buffer, avoiding internal fragmentation [Hazelwood and Smith 2002]. Both FLUSH and FIFO may prematurely discard a frag-
ment that will be needed again, i.e., they lead to non-compulsory misses in the F$. A non-compulsory miss requires to retranslate a fragment, increasing DBT overhead.

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. This issue is illustrated in Figure 3. The figure shows code before and after evicting fragment Fa. The branch in Fb is redirected to the target fragment iff the offset is small enough to be encoded in the signed 16-bit immediate field (i.e., ≤128K for PISA), otherwise the branch will be redirected to the trampoline and the trampoline will be overwritten with an absolute jump to the target fragment.

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 stop fragment formation when a control transfer instruction (CTI) is found, making the fragments Dynamic Basic Blocks (DBB). As demonstrated in Section 6, this configuration minimizes code duplication and is appropriate when memory is limited. Performance results are reported for three F$ sizes (64K, 32K and 16K) normalized to the performance with an unbounded F$.

The unbounded F$ baseline represents the “ideal” performance if no constraint was placed on the F$ size, i.e., the performance with an SPM at least as big as the F$ natural size.

Figure 4 shows the initial retranslation ratio – the number of fragments with an application address that has already been seen by the DBT, divided by the total number of fragments – for the MiBench applications with the three F$ sizes. In general, as F$ size decreases, the percentage of retranslated fragments increases. A higher retranslation ratio indicates that the F$ can hold less of the translated code working set. The 64K F$ is big enough to hold the translated code working set of several benchmarks (e.g., bitcount, qsort, stringsearch). With a 32K F$, adpcm.decode, adpcm.encode, blowfish.decode and blowfish.encode require some amount of retranslation with FLUSH, but none with FIFO. FIFO has the benefit of evicting only one fragment at a time instead of all fragments at once, at the cost of fixing backwards links targeting the evicted fragment. The 16K F$ is too small for some benchmarks, such as rijndael.decode and rijndael.encode, leading to a retranslation ratio greater than 99%. The translated code working set for ghostscript, ispell, lame, patricia and typeset does not fit even in 64K. For all sizes and eviction policies, the retranslation ratio for these benchmarks exceeds 99%.
In general, FIFO has better retranslation ratios than FLUSH. On average, retranslated fragments account for 27.11% (FC64K), 51.81% (FC32K) and 73.13% (FC16K) of all fragments with FLUSH. The average retranslation ratios with FIFO are 21.73% (FC64K), 42.02% (FC32K) and 66.97% (FC16K).

Figure 5 shows the performance of the MiBench programs relative to the unbounded F$ baseline. When the translated code working set of the benchmark fits in the F$ (indicated by the absence of retranslations, see Figure 4), the performance with the constrained F$ is equivalent to the performance with the unbounded F$. A small amount of retranslation leads to a small overhead, however. For instance, *adpcm.decode* and *adpcm.encode* achieve equivalent performance to the unbounded F$ with a 32K F$ with FIFO, but have a 2% overhead with a 32K F$ with FLUSH.

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$; it has 6% overhead with FLUSH and only 1% with FIFO. When running with a 32K F$, its slowdowns are 3.02x (FLUSH) and 11.88x (FIFO). However, with a 16K F$ it has slowdowns of 3127.97x (FLUSH) and 2605.35x (FIFO)! For *patricia*, significant slowdowns occur even with a 64K F$: 46.22x (FLUSH) and 60.11x (FIFO). The situation is especially bad for 16K: 4615.15x (FLUSH) and 4266.73x (FIFO)!

Different F$ sizes favor FIFO or FLUSH. *fft* does better with FLUSH for 32K F$, but for 16K FIFO is the better choice. *patricia* prefers FLUSH for 64K F$ and FIFO for 32K and 16K. 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, a F$ allocated to a small SPM can hold more fragments for a longer time before
Enabling Dynamic Binary Translation in Embedded Systems with Scratchpad Memory

Our techniques may increase the DBT's data footprint. Some of them need additional data structures to hold information in data memory instead of the F$. Keeping more fragments increases the size of the fragment map and related data structures. Reducing the DBT's data footprint is desirable for embedded systems to reduce main memory usage and potential data cache conflicts. However, as a step towards a complete solution for code and data, we focus on reducing the translated code footprint to make it fit in a small SPM and achieve acceptable performance.

4. TRANSLATED CODE COMPOSITION

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 [Bala et al. 2000; Hiser et al. 2006]. However, in embedded systems with limited F$ space (due to SPM), 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.

In this section, we propose a classification of the code generated by the DBT and study the composition of the F$ before enabling our techniques.

4.1. Instruction Categories

We classify the instructions emitted into the F$ according to their purpose. Figure 6 shows an example of untranslated code (left side) and its corresponding translated code (right side). The translated instructions are examples of each category below.

Fig. 5. Initial performance relative to unbounded F$
**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 6 (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 6, fragment F1 contains a series of native instructions (labelled “Native”).

**Trampoline instructions** are used to return control to the DBT 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 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
Enabling Dynamic Binary Translation in Embedded Systems with Scratchpad Memory

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 Code Composition and F$ Usage

We investigated how much of the code in the F$ corresponds to each instruction category. Figure 7 shows the amount of code generated per benchmark for each instruction category for an unbounded F$. When compared to Figure 5, the benchmarks without performance loss are those whose total amount of translated code fits in the bounded F$, such as \texttt{adpcm.decode}, \texttt{adpcm.encode} and \texttt{stringsearch} with 64K and 32K F$.

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

For several benchmarks, however, the code size for native and call emulation instructions sum to less than 64K (or, even 32K and 16K). For instance, in \texttt{basicmath} and \texttt{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 5. Thus, they could benefit from a reduction in the amount of control code.

Figure 8 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: 28.81% with FLUSH and 27.58% with FIFO. Trampoline instructions are the largest consumer of F$ space, averaging 56.46% with FLUSH and 59.20% with FIFO (due to eviction preparation). Code used to handle indirect branches averages 7.63% with FLUSH and 6.73% with FIFO. Prologue code averages 5.40% for FLUSH and 5.22% for FIFO. Call emulation averages 1.27% for both FLUSH and FIFO. Links average 0.42% with FLUSH and 0% with FIFO because we do not count links when they overwrite trampolines. For brevity, experimental results in Section 5 are shown for the benchmarks in Figure 8 for a 32K F$. In the final results, presented in Section 7, we show all the benchmarks with 16K, 32K and 64K F$ sizes with FLUSH and FIFO.

5. REDUCING CONTROL CODE SIZE

In this section, we describe and evaluate our techniques to minimize the amount 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. When appropriate, we show the relative percentage of code for each instruction category before and after we apply our techniques to see how well they reduce the total control code. We also study performance because a smaller code size can lead to fewer code cache evictions, but our techniques might incur some overhead. Thus, we have to consider performance, as well as relative code size, to understand their effectiveness. Our experimental configuration forms DBBs. The impact of forming larger code regions [Hiser et al. 2006] is discussed in Section 6. We initially focus on trampoline code, which is the greatest consumer of F$ space.

5.1. Direct Branch Trampolines

Trampolines help perform a context switch to the DBT. Their design is guided by the target architecture and the internal design of 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 [Scott et al. 2003]). To avoid unnecessary 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$. However, there are other unique opportunities to reduce the size of the trampoline.
We consider the alternative designs shown in Figure 9. They are ordered by trampoline size. A code reduction is achieved by moving the information associated with the trampoline into data memory to free F$ space, which increases the execution cost of the trampoline.

Design (a), “2-argument” (2-Arg), is 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. The registers are saved on top of the application stack. 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” (1-Arg), 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 ($&link). This approach inserts fewer instructions in the F$ 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” (Cont.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). This approach saves one instruction.

More F$ space can be saved by storing the trampoline data in main memory. For this purpose, a hash table indexed by trampoline address is used to store and recover the trampoline data. Design (d), “mapped data” (Map.Data), implements this approach. It trades data memory (an extra hash table) for fewer instructions. It also increases...
the execution cost of the trampoline since a hash table lookup must be performed on re-entering the builder.

Instead of having code to spill the link register in every trampoline before its value is overwritten (by the jump-and-link), design (e), “shadow link register” (ShadowLR), 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 ($sha$). Trampolines can then safely overwrite $ra$ since the re-entrance routine uses the value in $sha$ 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 by non-leaf procedures when passing the return address to callees and when recovering their own return address from the stack. Because the number of calls and returns from non-leaf procedures is usually much smaller than the number of CTIs requiring trampolines, this technique can be very effective.

For further space savings when implementing design (e), link register synchronization code is added to the beginning of callee fragments rather than to the fragments calling them. When translated, a direct call is transformed into a jump and call emulation instructions are inserted to set the value of the link register to the application return address. If the callee is not found in the F$, the reentry code accessed by the trampoline obtains the correct value of the link register by decoding the call emulation instructions and updates the shadow variable. When the target fragment of the call is translated, the code to update the shadow variable is added at its beginning. This strategy preserves correctness after fragment linking.

A DBT may perform register relocation [Luk et al. 2005; Nethercote and Seward 2007]. This allows reserving the link register for exclusive use by the DBT. Ensuring that the translated code does not redefine the link register makes the “shadow link register” unnecessary. A trampoline can then safely overwrite the link register. However, reducing the number of available registers may increase code size due to additional spill code. Since Strata currently does not perform register relocation, we do not evaluate this possibility.
Enabling Dynamic Binary Translation in Embedded Systems with Scratchpad Memory

Evaluation. Figure 10 shows the performance of the benchmarks for a 32K F$ relative to an unbounded F$. For some benchmarks, the initial gain obtained with “1-Argument” is significant: basicmath goes from 43.44x slowdown with FLUSH and 427.83x slowdown with FIFO to just 1.25x and 1.33x! Once close to the ideal, the improvements are less impressive: fft goes from initial slowdowns of 3.02x and 11.88x with FLUSH and FIFO to overheads of 4% and 2% for “1-Argument”. The other designs do not achieve further improvement. In benchmarks with high F$ pressure, the effect is progressive: ghostscript has slowdowns of 23.24x (2-Arg), 15.07x (1-Arg), 14.18x (Cont.Data), 13.41x (Map.Data) and 12.85x (ShadowLR) with FLUSH. patricia has slowdowns of 65.69x with FLUSH and 51.1x with FIFO when using ShadowLR. However, the other designs have slowdowns beyond 1000x. The greatest improvement overall is achieved with ShadowLR.

Figure 11 shows the 32K F$ utilization after applying “Shadow Link Register”. With both FLUSH and FIFO, native instructions now account for 59.2% of the F$ on average, while trampoline instructions are reduced to an average of 12.38% (FLUSH) and 13.65% (FIFO). We introduce a new category (“LR Sync”) to account for the code introduced to update the shadow variable; it averages 2.99% with FLUSH and 2.95% 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$ usage. Since “Shadow Link Register” (ShadowLR) has the best overall performance, we use it in the rest of the article.

5.2. Trampoline Placement

The arrangement of fragments and trampolines inside the F$ also plays a role in F$ pressure. A DBT often interleaves trampolines and fragments [Bala et al. 2000], as shown in Figure 12(a). Strata by default emits trampolines in this way, contiguous to the fragments that need them. When trampolines are bigger than links, this layout leads to unused F$ “holes” when a fragment is linked. After linking, the trampoline code is dead but its space, precious for a small F$, can not be reused since it is too small to hold a new fragment.

A key observation is that a trampoline is executed only once or never (if the translation of the target is requested by another trampoline, or the trampoline is flushed before execution). Thus, a trampoline can be deleted or reused by a different fragment after its original fragment is linked [Guha et al. 2007]. For that purpose, an alternative arrangement, shown in Figure 12(b) uses a trampoline pool. In this layout, the trampoline pool starts at one end of the F$, while the translated code starts at the other.

---

Fig. 11. Relative F$ usage after Shadow LR for 32K F$
end. The fragments and trampolines grow toward each other. A F$ overflow happens when the two meet [Guha et al. 2007]. Alternatively, the trampoline pool may be allocated in a separate, fixed-size F$ area [Hiser et al. 2006]. However, this alternative is less flexible and requires estimating the relative space requirements of fragments and trampolines. Thus, we do not consider it.

Although the “Shadow Link Register” design does not require a trampoline pool, the other designs can still benefit from it. How the pool is managed affects DBT overhead and F$ pressure. For example, a management scheme could trade space utilization for less translation time. We evaluate the following strategies:

(a) Delete the trampoline at the top of the pool when its associated fragment is linked [Guha et al. 2007]. The space occupied by the deleted trampoline can be reused for a new trampoline or fragment. The advantage to this technique is its simplicity and ability for the pool to shrink. Its disadvantage, however, is that any free space inside the pool can not be reclaimed.

(b) Maintain a trampoline Free List. This approach needs a list head pointer; a free trampoline itself embeds a pointer to the next free trampoline in the list. When a trampoline is needed and the free list is not empty, the last trampoline added to the list is reused. The pool is grown when all current trampolines are active, i.e., when the list is empty. No attempt is made to compact (shrink) the pool when it has free entries. The advantage to this scheme is its flexibility to reuse trampolines inside the pool. Its disadvantage is the cost to maintain the free list.

(c) Combined deletion and free list. In this scheme, the trampoline at the top of the pool is deleted when its fragment is linked. Otherwise, the free trampoline is added to the free list for future reuse. This scheme attempts to gain the advantages of both (a) and (b).

Evaluation. Figure 13 shows the slowdowns of the benchmarks without and with the trampoline pool, to compare the different strategies. We show the slowdowns with “Mapped Data” without the trampoline pool (Map.Data), with an unmanaged trampoline pool (Unmanaged), and with the three trampoline pool management strategies: Delete, FreeList and Combined. The slowdown with “Shadow Link Register” (ShadowLR), which does not need the trampoline pool, is also shown. The “Mapped Data” design is used because it is the one that achieves the best results while still having potential to benefit from the pool. We use only the FLUSH policy is used in this evaluation, since previous studies [Hiser et al. 2006; Guha et al. 2007] evict the trampoline pool along with all its associated fragments.
Using the trampoline pool may increase code footprint because an extra jump is necessary to link a fragment to its trampoline. Thus, an unmanaged trampoline pool rarely performs better than the interleaved layout. Unmanaged can have a much greater slowdown than Map.Data. For instance, \texttt{gsm.encode} has 46.94x slowdown with Map.Data and 251.28x slowdown with Unmanaged.

These results illustrate the need for a trampoline pool management strategy [Hiser et al. 2006; Guha et al. 2007]. The proposed trampoline pool management strategies almost always perform better than Unmanaged. One exception is \texttt{typeset}, which has 127.88x slowdown with Unmanaged and 128.13x slowdown with Delete. In this case, the extra effort necessary for Delete does not pay off with enough space savings to allow more fragments to fit into the F\$. Among the trampoline management policies, Delete is better than Unmanaged, but not as effective as FreeList. Combined does not offer a significant advantage over FreeList. The reason is that for most benchmarks the top trampoline is rarely executed. For instance, \texttt{patricia} has a slowdown of 1367.58x with Map.Data, which is increased with Unmanaged to 4135.86x. Although Delete helps to lower \texttt{patricia}'s slowdown to 4130.30x, FreeList and Combined do much better, with 69.17x slowdown.

There are cases in which the trampoline pool does not help. For instance, \texttt{lame} with Map.Data has 113.53x slowdown, but with FreeList (the best strategy) it has 120.31x slowdown. When a program already fits into the F\$, the results show that there is no major improvement obtained by using the pool. For instance, \texttt{basicmath} has 2% overhead with Map.Data and the managed trampoline pool (for all strategies).

When the “Shadow Link Register” (ShadowLR) design is used, trampolines are one instruction long and reclaiming (i.e., pooling) trampolines is unnecessary. Because trampoline pooling is unnecessary, ShadowLR could lead to better performance. We compare ShadowLR to the trampoline pool designs to explore this trade-off. We observe that the benchmarks with the greater slowdowns perform better with ShadowLR than with the trampoline pool. For instance, \texttt{patricia} with FreeList and Combined has 69.17x slowdown, which is not as good as its 65.69x slowdown with ShadowLR. There are several cases in which the trampoline pool with the best management strategy outperforms ShadowLR. For example, for \texttt{gsm.encode} the pool with FreeList and Combined has a 7.65x slowdown, which is better than the 8.10x slowdown with ShadowLR.

On average, the trampoline pool managed with FreeList has a 25.13x slowdown, which is the same average slowdown with Combined. This slowdown is better than the 132.06x slowdown achieved with Map.Data, indicating that a trampoline pool with FreeList management should be used if fragment linking leaves unused holes in the F\$. However, the average slowdown with ShadowLR, which solves the problem of unused holes, is 24.73x, outperforming the trampoline pool. Although the use of a trampo-
line pool can be a good strategy, ShadowLR proves to be more effective in reducing F$ pressure. Furthermore, when fragments and trampolines grow towards each other, the implementation of F$ management policies such as FIFO becomes more complicated. For this reason, we use ShadowLR for the rest of the paper.

5.3. Indirect CTI Handling

After reducing trampoline size, indirect CTIs become a more important source of F$ pressure. In Figure 11, the amount of code generated for indirect CTI 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 CTI 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 translated fragment 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) [Hiserr et al. 2011]. The IBTC is a small, direct-mapped table that associates indirect CTI target addresses to their F$ locations. The table is allocated in main memory, but the code that accesses it is emitted into the F$.

Figure 14(a) shows the code generated in the F$ to access the IBTC for each indirect branch. 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 the link register ($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$. Our approach trades dynamic instruction count for more compact code with a single “Out-Of-Line IBTC Lookup” (OOL-Lookup) as shown in Figure 14(b). The shared out-of-line lookup code is similar to a function call — arguments are passed to the code to indicate the requested application address and a pointer to the fragment map’s record of the fragment with the indirect CTI (to pass to the builder on a miss). The out-of-line lookup code is emitted in the F$ during initialization.

An equivalent of the “contiguous data” trampoline for indirect CTIs can also be implemented, as shown in Figure 14(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 indirect CTIs that use the same register. As shown in Figure 14(d), the code generated for each indirect CTI 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 target registers are discovered by the DBT.
Enabling Dynamic Binary Translation in Embedded Systems with Scratchpad Memory

A final alternative is to implement “Shared Target Register Copies” as part of Strata's code in ROM (STRC-ROM), instead of emitting the code into the F$. In this way, it is possible to have shared transfer routines for all registers, while the on-demand allocation with FIFO may need to revert to Contiguous Data due to lack of room in the F$. On the other hand, the address of the table cannot be loaded as a constant in the code. It must be obtained from a variable in memory.

**Evaluation.** Figure 15 shows the performance of the benchmarks with the proposed IBTC lookups. The ShadowLR results (from Section 5.1) use the Inline IBTC Lookup approach. Progressive improvements are obtained with each scheme.

The greatest improvement is obtained for *patricia*: its 65.69x slowdown with FLUSH after ShadowLR is reduced to 1.83x when the IBTC lookup code is moved out-of-line. Further reductions are obtained by moving the data after the code and factoring out the target register copy: 1.37x (CDI), 1.22x (STRC) and 1.23x (STRC-ROM). *ghostscript* shows more steady slowdown reductions. For instance, *ghostscript* slowdowns with FLUSH are 12.85x (ShadowLR), 11.80x (OOL-Lkup), 11.61x (CDI), 11.22x (STRC) and 10.92x (STRC-ROM).

The benchmarks that already fit, however, suffer some performance degradation due to the increased number of jumps. The 5% overhead of *basicmath* with FIFO and Shad-
A:20 J. Baiocchi et al.

Fig. 15. Performance of IBTC lookup placements

Fig. 16. Relative F$-32K usage after STRC

owLR is increased to 7% (OOL-Lkup and CDI), 8% (STRC) and 11% (STRC-ROM). Similar overheads are obtained for basicmath with FLUSH.

In the rest of the article, we use “Shared Target Register Copies” implemented as part of Strata (STRC-ROM) because it achieves the greatest improvement. Figure 16 shows the 32K F$ utilization after STRC-ROM. Native instructions now account for 68.87% with FLUSH and for 68.78% with FIFO (an increase of about 10% after enabling only ShadowLR). Indirect CTI handling code is reduced to an average of 2.29% with FLUSH and 2.26% with FIFO.

5.4. Prologue Elimination

After reducing the indirect branch handling code, fragment prologue instructions now account for an average 9% of the F$. The exec routine in Strata/PISA, shown in Figure 17(a), uses an indirect jump to transfer control to the fragment whose address is stored in the link register ($ra). Thus, the fragment prologue must restore $ra. On an IBTC hit, restoring $ra is also done by the fragment prologue.

In this section we show how to eliminate the prologue code, and an optimization (Bottom Jump Eliding) enabled only after no prologue is needed. On some architectures, like ARM [Moore et al. 2009], it is possible to transfer control from the DBT to a fragment without adding prologue code. In that case, prologue elimination is unnecessary but Bottom Jump Eliding is still applicable.

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 17(b). As shown, instead of ending with an indirect jump, the routine that returns control to the F$ (sm_exec) 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, SPM addresses are not cached and a data write immediately modifies the SPM. As a result, synchronization is not needed and self-modifying code is inexpensive. Because Strata is in ROM, our implementation initially emits the “return routine” into the F$ on start-up. The code for an IBTC hit must also be modified: to go to the target fragment, a direct jump is overwritten with the target fragment address found in the IBTC.

Another alternative is shown in Figure 17(c), SMCT-ROM. In this case, the code that overwrites the jump is part of Strata (stored in ROM) so only 2 instructions are used in the F$ to restore the link register and jump to the fragment through the modified jump. The cost is an extra indirection: the address of the smaller “return routine” in the F$ is stored in a variable, from which it is loaded to the link register ($ra).

**Bottom Jump Eliding.** After eliminating the prologue, there is no need to skip it when fragments are linked. If a trampoline is the last piece of code emitted into the F$, instead of patching it with a jump to the new fragment, the trampoline can be overwritten with the fragment code. “Bottom Jump Eliding” (BJE) implements this idea. Figure 18 shows the translation of direct calls and branches before (middle) and after (bottom) modifying the fragment to let it fall-through into its successor fragment. For unconditional jumps, calls and taken conditional branches, as shown in Figure 18(a)-(b), it is enough to start the new fragment in place of the trampoline. For not taken conditional branches, the branch condition must be negated to maintain correctness, and the trampoline for the original not-taken target modified to request the translation of the original taken target instead.
When combining BJE with F$ management policies (e.g., LRU) that may evict the successor fragment but not the fragment that flows into it, the trampoline must be regenerated. To enable BJE, the DBT must guarantee that the evicted fragment is bigger than a jump when using the trampoline pool or bigger than a trampoline when interleaving fragments and trampolines. With single-instruction trampolines, all fragments satisfy this condition.

**Evaluation.** Figure 19 shows the slowdown of the benchmarks without (STRC-ROM) and with prologue elimination (SMCT and SMCT-ROM). The results after enabling Bottom Jump Eliding with SMCT-ROM are also shown (BJE). High-pressure benchmarks show significant improvements: *ghostscript* with FLUSH goes from a 10.92x slowdown (STRC-ROM) to 8.9x (SMCT and SMCT-ROM) and 6.65x with BJE. With FIFO, *ghostscript* slowdown is reduced from 8.24x (STRC-ROM) to 6.41x (SMCT and SMCT-ROM) and 4.67x (BJE).
In some benchmarks, SMCT increases the slowdown and BJE helps reduce it again. This situation happens for pgp.encode with FLUSH. It has a 1.57x slowdown with STRC-ROM, which increases to 1.61x with SMCT and SMCT-ROM. With BJE, the slowdown is decreased to 1.49x. Performance differences between SMCT and SMCT-ROM are only noticeable for the benchmarks with very high slowdowns, e.g., typeset has slowdowns of 50.82x and 50.26x (FLUSH).

Figure 20 shows the F$ usage 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). In the rest of the article, we use the “self-modifying control transfer” (in ROM) combined with “bottom jump eliding”, since this combination achieves the better performance improvement. We next study how different fragment formation alternatives impact the amount of generated code.

6. IMPACT OF FRAGMENT FORMATION POLICIES

A DBT translates a program on demand following the execution path. It may choose to terminate a fragment when a CTI is found and translated. In this section, we study how different choices affect code footprint, and ultimately, performance.

6.1. Fragment Construction Alternatives and Problems

When an instruction is fetched, the DBT has several options on how to handle it. The choice of options for a particular DBT affects code footprint due to code duplication and translation of code that is never executed.

When fragment formation is continued at a direct CTI, code duplication can happen. If the target of the direct CTI was already in the F$, it will be duplicated (inlined) in the currently translated fragment. In general-purpose systems, this reduces the instruction count and helps instruction locality. For a bounded F$, it increases the translated code footprint and causes extra overflows.

Other configuration choices lead to dead code translation. This situation happens when a target of a conditional CTI is speculatively translated but the condition to execute it never occurs before a F$ overflow or the end of execution. Continuing fragment formation at the return address of a call has a similar problem. In this case, the called procedure may not fit in the F$, which triggers a flush before the code after the return is executed.

Table II summarizes the options for continuing or ending a fragment based on the type of instruction fetched by the DBT:
Table II. Fragment construction alternatives

<table>
<thead>
<tr>
<th></th>
<th>Stop Alternatives</th>
<th>Continue Alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-CTI</td>
<td>Never</td>
<td>- Next Instruction</td>
</tr>
<tr>
<td>Unconditional Jump</td>
<td>Never</td>
<td>- Target (Elide)</td>
</tr>
<tr>
<td></td>
<td>- If Target Cached</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Always</td>
<td></td>
</tr>
<tr>
<td>Conditional Branch</td>
<td>Never</td>
<td>- Fallthrough</td>
</tr>
<tr>
<td></td>
<td>- If Target Cached</td>
<td>- Target</td>
</tr>
<tr>
<td></td>
<td>- If Fallthrough Cached</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- If Either Cached</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- If Both Cached</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Always</td>
<td></td>
</tr>
<tr>
<td>Direct Call</td>
<td>- Never</td>
<td>- Target (Inline)</td>
</tr>
<tr>
<td></td>
<td>- If Target Cached</td>
<td>- Return Address</td>
</tr>
<tr>
<td></td>
<td>- If Ret.Addr. Cached</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- If Either Cached</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- If BothCached</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Always</td>
<td></td>
</tr>
<tr>
<td>Indirect</td>
<td>- Always</td>
<td></td>
</tr>
</tbody>
</table>

Table III. Evaluated DBT Fragment Formation Strategies

<table>
<thead>
<tr>
<th></th>
<th>DBB</th>
<th>Original Strata</th>
<th>Optimized Strata</th>
<th>LRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconditional Jump</td>
<td>Always stop</td>
<td>Always continue (fallthrough)</td>
<td>If target cached, stop, else elide</td>
<td>If target cached, stop, else elide</td>
</tr>
<tr>
<td>Conditional Branch</td>
<td>Always stop</td>
<td>Always stop</td>
<td>Always continue (fallthrough)</td>
<td>Always continue (fallthrough)</td>
</tr>
<tr>
<td>Direct Call</td>
<td>Always stop</td>
<td>Always inline</td>
<td>Always stop</td>
<td>Always continue (return address)</td>
</tr>
</tbody>
</table>

— An instruction that is not used for control transfers (non-CTI) is never an ending point. Translation continues with the next instruction.
— Unconditional jumps have a statically known target that is always taken. The fragment can be terminated or continued at the target address, eliding the CTI.
— Conditional branches define two targets, which are both statically known. The fragment can be ended at the conditional or, alternatively, translation may continue speculatively down one path.
— Direct calls are similar to conditional CTIs – they have a target and a fallthrough (the return address). However, both paths are eventually executed. It is possible to emit the call's target in a separate fragment or to partially inline it. The return address could also be used to continue the fragment, if the target of the call is not inlined.
— An indirect always ends a fragment because its target address is unknown during translation.

The decision to end a fragment on a direct CTI can be taken absolutely (always end or always continue) or it can consider whether the target fragment is in the $F$.

6.2. Evaluation of State-of-the-Art Fragment Formation Strategies

We explore how fragment formation strategies affect performance and code footprint. Since exploring all possible combinations would require too much space, we chose combinations used by state-of-the-art DBTs. We explore the four configurations described in Table III:

— Dynamic Basic Blocks (DBB) [Smith and Nair 2005] is a configuration where fragments are terminated at every CTI, so they correspond to dynamically constructed basic blocks. This is the configuration used for the previous experiments.
— **Original Strata** is Strata’s original configuration [Scott et al. 2003]. It is also similar to the configuration used to form “basic blocks” in DynamoRIO [Bruening 2004]. It stops fragments only when the target of a CTI is unknown at runtime.

— **Optimized Strata** is a configuration optimized to minimize dynamic instruction count with an unbounded F$ [Hiser et al. 2006].

— **Least Redundant Effort** (LRE) is the configuration used by HDTrans [Sridhar et al. 2006], which attempts to maximize reuse of translation effort.

For the evaluation, we enabled the best techniques to reduce control code size described earlier in this article. To handle F$ overflows we use the FLUSH policy, but similar trends are expected with FIFO or other code cache management policies. Figure 21 shows the slowdown for the last three configurations, relative to DBB, with a 32K F$.

The **Original Strata** configuration sometimes achieves performance close to DBB, but it is generally not as good. On average, it adds an overhead of 19%. In some cases it is almost equivalent (≤1% overhead) to DBB, e.g., `bitcount` and `qsort`. For benchmarks like `ispell` and `typeset`, the slowdowns over DBB are 3.55x and 2.35x. The **Optimized Strata** configuration significantly increases the pressure on the F$, leading to performance degradation. `bitcount` has an overhead of 5%. Benchmarks like `gsm.encode` and `patricia` do not fit into the F$, and have slowdowns of 733.11x and 8523.77x! **LRE** leads to similar problems. For instance, `fft` has a slowdown of 715.02x and `gsm.encode` has a slowdown of 1009.56x relative to the performance achieved with DBB.

The increased F$ pressure can be attributed to the increase in code duplication and the enabling of speculative translation. The percentage of duplicated application instruction fetches is shown in Figure 22 for the four policies with a 32K F$. These
results consider duplication due both to fragment formation policy and re-translation of prematurely evicted fragments. Although DBB has some duplication, the duplication is increased significantly by employing the other policies. For instance, `bitcount` needs 6.2% duplicated fetches with DBB, 11.05% with Original Strata, 35.98% with Optimized Strata and 39.34% with LRE. The effect on `fft` is more impressive. The percentages of duplicated fetches for `fft` are: 11.28% (DBB), 34.66% (Original Strata), 57.27% (Optimized Strata) and 99.98% (LRE).

The percentage of dead code is shown in Figure 23. It is a measure of wasted translation effort, i.e., translating instructions that are evicted prior to their execution. Some of this effort is unavoidable, such as trampolines that must be emitted but are never taken. DBB is again the best policy. For instance, in `bitcount`, 8.13% of the code generated with DBB is never executed, while the percentages with the other policies are 8.09% (Original Strata), 49.74% (Optimized Strata) and 53.97% (LRE). On average, 7.04% of the code emitted by the DBT when configured to form DBBs is dead, for Original Strata the percentage of dead code is 7.00%, for Optimized Strata it is 44.66% and for LRE it is 57.29%.

These results confirm that forming Dynamic Basic Blocks (DBB) is the most appropriate fragment formation strategy for embedded systems with a bounded F$. Thus, we use DBB in our overall evaluation.
7. OVERALL RESULT
In this section we discuss code footprint reduction and performance results for all benchmarks when forming DBBs and applying our best control code size reduction techniques.

Figure 24 shows the slowdowns of all benchmarks relative 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 4702.09x slowdown for a 16K F$ and 43.44x for a 32K F$ with FLUSH. Our techniques reduce these slowdowns to only 1.51x (16K) and 1.13x (32K). The final slowdown for a 64K F$ is 1.12x. It was initially only 1.01x, which indicates a performance cost associated with our techniques that is not amortized when the translated code working set fits in the F$.

patricia has an impressive improvement. For a 32K F$, its initial slowdowns were 4605.60x with FLUSH and 4018.44x with FIFO. Our techniques reduce both of them to 1.08x. dijkstra in a 16K F$ is another example. Our techniques make its final performance equivalent to the unbounded F$ after an initial 17.02x slowdown.

Although our techniques reduce slowdown by improving the usage of the small F$ in SPM, there are situations where the slowdowns can not be overcome. For instance, in a 32K F$ with FLUSH, the initial slowdown of 117.70x for lame is barely reduced to 116.22x. Our techniques help somewhat, but can not fully overcome the performance degradation when the application code working set does not fit in the F$. This can be observed in Figure 25.

Figure 25 shows the relative reduction in the amount of control and application code generated for a 32K F$ (with FLUSH and FIFO) when our techniques are applied. For lame, control code is reduced by 88.88%, but application code is reduced by only 1.50%. Thus, performance is barely improved. On the other hand, patricia has reductions of
almost 100% in both control and application code, i.e., the translated code now fits, which explains its impressive performance improvement.
Our techniques reduce code size and lead to a performance improvement regardless of the F$ management technique. Figure 26 shows the performance improvement after the application of our techniques for the 64K, 32K and 16K F$. When applying our techniques with FLUSH, we obtain speedups of 5.13x for a 64K F$, 150.94x for a 32K F$ and 284.83x for a 16K F$ relative to the initial performance. The speedups we obtain with FIFO are 5.65x (64K), 144.05x (32K) and 264.40x (16K).

8. LIMITATIONS

The techniques in this paper reduce the memory requirements for translated code, enabling the use of a DBT in embedded systems with scratchpad. Although our techniques increase the applicability of DBT, there are limitations to the approaches.

The first limitation is our techniques have been devised for single-threaded applications and primarily target private fragment caches (i.e., each thread has its own F$). A private F$ is appropriate for situations where threads do not share code paths. In embedded systems, particularly lower-end ones, it is likely that threads do different tasks, and thus, may not share code paths. A private F$ in this situation is appropriate, and our techniques can be directly applied. However, when threads share code paths, a shared F$ will perform better than multiple private F$’s [Bruening and Kiriansky 2008]. Further, a shared F$ will have a smaller code footprint due to less duplication, allowing it to possibly fit in the SPM without our techniques (if the working set size permits). These types of applications are more common in higher-end systems, particularly servers (e.g., Apache spawns threads to do the same work for different web requests). We expect that many of our techniques can be extended to a shared F$ with appropriate synchronization and selective control code duplication (e.g., give each thread its own private jump instruction slot for modification by a self-modifying control transfer). The extra synchronization may impose more performance and space overhead. The trade-offs and techniques to allow a shared F$ to fit in SPM remain an interesting avenue for further study.

A second limitation is our techniques depend on the relationship of the DBT to the threading sub-system. If the DBT operates on top of threading, synchronization is needed to prevent multiple threads from editing the shared F$ [Bruening et al. 2006]. In this situation, it is unsafe to use the application stack as temporary storage. Some of our techniques (i.e., single-instruction trampolines, bottom jump eliding) are unsafe if a thread using them can be pre-empted. However, if the DBT operates below the threading system, or if threading is provided by the DBT (i.e., “green threads”), our techniques can be used without major changes, since the multi-threaded program will appear as a single instruction stream to the DBT.

A third limitation is our techniques were implemented and evaluated for a MIPS-like ISA. The techniques can be extended to other common RISC ISAs, such as ARM. Each target ISA has its own nuances that must be handled by a DBT [Hiser et al. 2011; Moore et al. 2009]. We expect some changes to the instruction sequences presented in the paper will be needed, but the concepts apply to other ISAs. Although Intel x86 is less likely to be used in the low-end resource-constrained SoCs targeted by our techniques, the approaches can be used for x86. To support x86 requires more changes than ISAs which are closer to MIPS. The principal issue is the benefit of the techniques will likely be significantly different. Self-modifying control transfers may harm performance, if there is automatic synchronization of SPM, similar to data and instruction caches for x86. Similarly, eliminating prologue instructions is unnecessary, as x86 allows an indirect jump from a memory location. A different indirect handling mechanism, a sieve and return address cache, is a better choice than the IBTC [Sridhar et al. 2006; Hiser et al. 2011]. The concept of factoring the IBTC code can be applied to
move the sieve’s hash buckets out of the F$. Finally, the single instruction trampoline is unsafe on x86, as a call modifies the stack in memory.

A fourth limitation is our approaches do not address DBT data structure size. In some cases, our techniques require additional records and lookup tables. As more fragments fit in the F$, the amount of auxiliary DBT data structures is increased. However, the F$ management policy also influences bookkeeping requirements. If single-fragment deletion is not supported, fragment records are not needed for every fragment. We implement an optimization to reclaim storage for trampoline records as soon as trampolines are linked to their targets. Link records are also reclaimed if the target fragment is not going to be deleted before the source fragment, according to the F$ management policy. Performance and memory requirements can also be balanced with clever F$ management, such as selective flushing [Guha et al. 2010].

A fifth limitation is our techniques have not been evaluated in the presence dynamically linked libraries (DLLs) and self-modifying code Bruening and Amarasinghe [2005]. The experimental work in this paper does not deal with these issues: SimpleScalar uses statically linked, single-threaded binaries without self-modifying code. Potential changes to the original code increase bookkeeping requirements as any fragment might require invalidation to maintain F$ consistency. Bottom jump eliding must be undone if the target fragment is deleted, and the remaining hole in the F$ might be difficult to re-use for a new fragment.

The last limitation is our techniques reduce F$ space requirements but increase execution cost of an individual trampoline or IBTC lookup, since more CTIs are used. In applications where the translated code working set already fits in the F$, our techniques may increase overhead and should be disabled. The techniques are most beneficial when the translated code working set initially exceeds SPM capacity. With a larger SPM, our techniques might not be needed. However, a more complex application with larger code size might again require the techniques to achieve good performance.

9. RELATED WORK

SPM has been proposed as an alternative or complement to hardware-controlled caches for embedded systems. Banakar et al. [2002] show that SPM uses less chip-area and energy than an equivalent hardware cache. The SPM must be managed by software, and several compiler-based approaches have been devised to allocate code and data to the SPM. For example, Verma and Marwedel [2006] and Udayakumaran et al. [2006] have devised this kind of approaches, which require knowing the size of the SPM at compile time.

An alternative approach is to emulate the behavior of a hardware cache using the SPM, known as “software caching”. Miller and Agarwal [2006] develop a software instruction caching system for processors with SPM and no hardware instruction cache. Their system is not a DBT, since it uses a binary rewriter to statically form cache blocks of fixed size and inject control code prior to the program’s execution. Our work shows that a DBT with a bounded F$ allocated to SPM effectively provides this functionality. With the use of a DBT no binary modifications are required before execution, and other services provided by the DBT can also be enabled.

Several studies address diverse F$ management issues, mostly for general-purpose systems. Bala et al. [2000] flush the trace cache of a dynamic optimizer on demand or when detecting program phase changes. Hazelwood and Smith [2002] explore different eviction policies and conclude that FIFO has only half the miss rate of FLUSH. Later [2006], they show that mid-grained evictions scale better than FLUSH and FIFO. 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. [2008]
Enabling Dynamic Binary Translation in Embedded Systems with Scratchpad Memory

adapt this generational scheme to embedded systems, with the goal of reducing the dynamic size of the F$. These techniques focus on choosing which fragments to discard and which fragments to keep in the F$. The techniques in this paper focus instead on reducing the size of the translated code. Thus, they are complementary and independent of eviction techniques.

In previous work [2007], we showed how to increase the effective capacity of a small F$ in SPM by creating a transient compressed victim F$. This technique reduces the retranslation cost of accessing Flash memory. “Pinning” decompressed fragments in the F$ can avoid repeated compressions and decompressions [2007].

Bruening [2004] shows that allocating trampolines in a separate memory region (a “trampoline pool”) reduces the dynamic size of the F$. He also suggests deleting the trampolines after fragment linking and observes that about half of them can be deleted. He shows breakdowns of the source of code expansion similar to those presented in this paper. In general-purpose systems, the separation of fragments and trampolines reduces pressure on the hardware instruction cache. Hiser et al. [2006] show that over 18% of misses are eliminated when using a trampoline pool.

Reducing the code size of trampolines has been studied by Guha et al. [2007]. Their setup uses a F$ divided into several units (for mid-grained evictions), each with a trampoline pool that grows towards the fragments. They propose reducing the trampoline size and deleting free trampolines on top of the trampoline pool in the current unit. They also unify trampolines that request the same address. They discuss the FreeList idea, but choose not to implement it. Unlike our implementation, their FreeList proposal does not reuse free trampolines themselves to embed next node pointers. Their optimizations reduce the relative size of trampolines from 66.7% to 41.4%, and obtain a 1.5% performance improvement for an unbounded F$, relative to a F$ with an unmanaged trampoline pool. Our work shows that reducing trampoline size to one instruction (“shadow link register”) makes a trampoline pool unnecessary while achieving similar or better performance.

The trampoline designs in Section 5.1 are conceptually similar to some trampoline designs in existing DBTs. To our knowledge, comparing different trampoline design alternatives has not previously been done, since designers typically propose one particular trampoline design for their DBT. Design (a) is Strata’s default [Scott et al. 2003]. Design (b) is similar to the trampolines generated by DynamoRIO [Bruening 2004] and Pin [Luk et al. 2005]. Design (c) is similar to Dynamo’s trampolines [Bala et al. 2000]. Guha et al. [2007] also propose an idea similar to Design (c) for reducing the size of trampolines. A similar idea to Design (e) is used by Miller and Agarwal [2006]. Their binary rewriter inserts code to update the “shadow link register” after each definition of the link register (including call sites) and to load its value before each use. Instead, our DBT updates the “shadow link register” at the beginning of each callee. We have no need to instrument uses, because the proper value of the link register is restored on return from the DBT. Moore et al. [2009] propose a trampoline design to improve hardware cache performance in the ARM architecture. The trampoline sets an argument register with an index into a table. The table, in data memory, holds the arguments to pass to the builder.

Hiser et al. [2011] study several DBT’s indirect branch handling techniques across platforms. They found the IBTC to be the most useful technique, but that the placement of the lookup code has little importance in general-purpose systems. Bruening [2004] observes little performance improvement when inlining the IBTC lookup, which increases the size of translated code. Our work shows that, in embedded systems with a small F$, aggressive factorization of the lookup code can benefit performance due to space savings.
Bruening [2004] also studies the use of a fragment prologue to restore a scratch register when entering a fragment after an IBTC hit. He suggests eliminating the prologue if the scratch register is dead in the fragment. Our self-modifying context switch eliminates the need for a fragment prologue.

Bruening [2004] has studied the performance impact of code duplication due to eliding unconditional CTIs. A more general study on fragment formation has been done by Hiser et al. [2006], who select the options that achieve the best performance in a general-purpose system with an unbounded F$. We show that those choices ("Optimized Strata") are not appropriate for a bounded F$ due to an excessive amount of duplicated and dead code.

10. CONCLUSION
In this article, we 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 F$ for an embedded system with on-chip scratchpad memory. We also showed that Dynamic Basic Blocks (DBB) are preferred as translation units for embedded systems with the F$ in SPM since they introduce the least amount of duplicated and dead code. In general, when our approaches are enabled, the translated code working set of a program running under DBT control is much more likely to fit in a constrained F$.

REFERENCES


In Design, Automation Test in Europe Conference Exhibition (DATE), 2011, 1–6.


Enabling Dynamic Binary Translation in Embedded Systems with Scratchpad Memory


Received February 2009; revised October 2011; accepted December 2011