Fragment Cache Management for Dynamic Binary Translators in Embedded Systems with Scratchpad

José Baiocchi†, Bruce R. Childers†, Jack W. Davidson‡, Jason D. Hiser‡, Jonathan Misurda†

†Department of Computer Science
University of Pittsburgh
{baiocchi, childers, jmisurda}@cs.pitt.edu

‡Department of Computer Science
University of Virginia
{jwd, hiser}@cs.virginia.edu

ABSTRACT

Dynamic binary translation (DBT) has been used to achieve numerous goals (e.g., better performance) for general-purpose computers. Recently, DBT has also attracted attention for embedded systems. However, a challenge to DBT in this domain is stringent constraints on memory and performance. The translated code buffer used by DBT may occupy too much memory space. This paper proposes novel schemes to manage this buffer with scratchpad memory. We use footprint reduction to minimize the space needed by the translated code, victim compression to reduce the cost of retranslating previously seen code, and fragment pinning to avoid evicting needed code. We comprehensively evaluate our techniques to demonstrate their effectiveness.

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

Algorithms, Measurement, Performance, Design, Languages.

Keywords

Dynamic Binary Translation, Embedded Systems, Scratchpad.

1. INTRODUCTION

Dynamic binary translation (DBT) has gained much attention as a powerful technique for constructing adaptive software [2, 3, 6, 21, 24]. DBT has led to new software capabilities, such as resource virtualization, intrusion detection, performance improvement, and instruction set migration. Although DBT has been widely applied to general-purpose systems, recent work has shown several uses of DBT for embedded systems, including power management [26], security [17, 22], software caches [19], instruction set translation [6] and memory management [23, 27].

While DBT is beneficial in embedded systems, the use of the technology has been limited in this domain due to tight constraints on memory and performance. In particular, DBT systems typically employ a software-managed memory buffer, called a *fragment cache (F$)*, to hold blocks of dynamically translated instructions (called *fragments*). To ensure low runtime overhead, the fragment cache is relatively large to hold an application's translated code working set, which avoids unnecessarily re-translating previously seen code. A typical F$ can be several megabytes in size, which may not fit in an embedded system's limited memory resources.

Many embedded systems, particularly those based on a system-on-a-chip (SoC), have a small on-chip *scratchpad* memory (SPM). The SPM may hold data, or possibly instructions. The advantage to the scratchpad over external memory is its fast access time and low power consumption. A typical SoC also employs Flash memory as permanent storage to hold application code. The Flash memory is unfortunately often quite slow and power hungry. When a program is executed, it is loaded into external main memory to minimize the costs associated with the Flash memory.

Due to its fast access and low power consumption, the scratchpad is potentially an appropriate resource to hold translated code in a DBT system (i.e., the fragment cache). However, the scratchpad is much smaller than the amount of space normally allocated to the fragment cache. If the F$ size is simply set to the scratchpad size, then the working set of the translated application code is unlikely to fit. As a result, there will be many off-chip accesses to re-translate previously encountered instructions. The high cost of these accesses negates the benefit of the scratchpad's fast access (and low power consumption) for the fragment cache.

In this paper, we propose a new approach to managing the F$ for embedded systems with SPM, external memory and Flash storage. The approach applies three novel management strategies to minimize the number of off-chip accesses to fetch and translate the program. First, the approach uses *footprint reduction* to minimize the amount of code that is generated by the dynamic translator to remain in control of the application. Next, the approach uses *victim compression* to reduce the cost of re-translating application code that may be evicted when the working set does not fit in the F$. Lastly, the approach uses *fragment pinning* to avoid evicting frequently executed fragments. We show that our techniques are effective and allow the fragment cache to fit in the scratchpad.

This paper makes several contributions, including:

- Footprint reduction to minimize code expansion from the dynamic translator;
- Victim compression to bypass re-fetching and re-translating the code when it is needed again;
- Fragment pinning to avoid unnecessarily evicting important and often needed code; and,
- A thorough evaluation of our techniques in a simulated SoC with scratchpad, SDRAM and Flash memories.

The paper is organized as follows. Section 2 describes the systems targeted by our techniques and Section 3 investigates how the F$ affects performance. Section 4 presents our techniques for the F$ and Section 5 gives the overall improvement with these techniques. Section 6 describes related work and Section 7 concludes.

2. TARGET SYSTEM

Figure 1 shows a canonical embedded system; this device is a single chip with a processor, L1 instruction (I-cache) and data (D-cache) caches, an application-specific integrated circuit (ASIC), a scratchpad (implemented as SRAM), ROM (implemented as a small on-device Flash memory), a controller for external Flash memory, a controller for external main memory (implemented as SDRAM) and off-chip I/O channels. The figure shows SDRAM and Flash memories, which are external to the device. The SDRAM is main memory and holds application code and data. The Flash memory is managed by the operating system (OS); it holds user files, including application binary images.

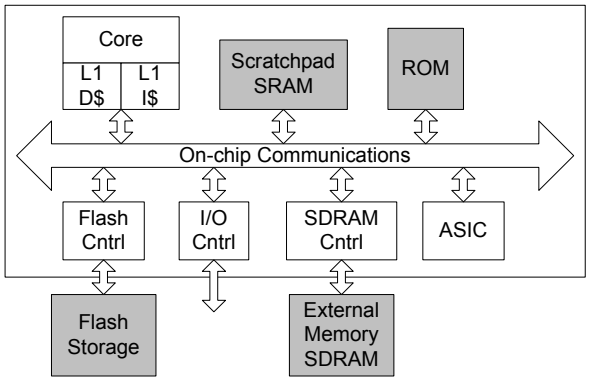

Figure 1: Example target embedded system.

Depending on the design, the boot-up and OS code may be in the ROM. On power-up, this code is loaded into external main memory and executed. An user application is also loaded from Flash memory into the external memory, where it will execute. Such *shadow memory* is common due to the Flash memory's high latency and power cost. For example, Microsoft PocketPC computers shadow applications. The Flash may have a raw access latency of a hundred clock cycles or more (without OS overhead), while the external memory may have an access latency of less than ten clock cycles. The processor may also have L1 data and instruction caches to reduce the cost of accessing external memory.

The SPM is part of the address space. It is typically small, say 16 to 64 kilobytes, because it is on-chip. As a result, it has a fast access latency. For example, it might take one to three clock cycles to fetch a word from the scratchpad. The scratchpad is managed by software—e.g., one approach uses the compiler to allocate hot code to it [7, 8, 20, 25]. In designs with scratchpads that have a single clock cycle access latency, the L1 instruction cache may not be included (i.e., its chip area is devoted to the scratchpad).

Now, suppose we want to use DBT in this environment. The dynamic translator can be kept in ROM as part of the system code, from which it will be copied into main memory, along with the OS at boot-up. The role of the DBT system is to translate the application code for some purpose, such as security [17] or power management [26]. When an application is initiated, the translator fetches the application code, one piece at a time, from Flash storage. It will translate the code and put the translated code into the fragment cache. The translated code executes from the F$. The dynamic translator inherently does *incremental loading*. It brings in pieces of code on-demand: The application is translated and written into the F$, which effectively serves as shadow memory.

The question is where to place the fragment cache: Should it be in external main memory or the scratchpad? There is a trade off between these choices. In the first one, the F$ is large with a slow access time because it is in main memory, while in the second one, the F$ is small with a fast access time because it is in scratchpad. The advantage to a large fragment cache is that more of the application's code working set can be captured, and as a result, there will be few, if any, evictions from the fragment cache. If the F$ is allocated to scratchpad, it will have a faster access time but possibly more evictions due to its small size.

We investigate this trade off and develop techniques to minimize the number of flushes and the penalty of fetching previously translated code (i.e., evicted code) into the F$. We aim to get the best of both approaches: A F$ with an effectively large size and a small access latency by executing translated code from the scratchpad.

3. IMPACT OF MEMORY CONSTRAINT

To motivate our techniques, we first investigate how constraining F$ size affects the performance. In particular, we study how program performance is affected when the fragment cache is allocated to the scratchpad, which limits its capacity to the SPM size.

3.1 Experimental Methodology

For this study, we use the DBT system, Strata [21], which we retargeted to SimpleScalar PISA [1]. Strata is a highly configurable and retargetable binary translator. The techniques described in the paper are implemented in Strata to accurately account for their overhead and impact on performance.

SimpleScalar was extended with Flash and SPM. It was configured to model the Intel/Marvell 624 MHz XScale PXA-270 SoC, which we augmented with SPM, SDRAM, and NOR Flash [14]. The PXA-270 is used in devices such as the Dell Axim x50v PocketPC. Details about the simulated processor, SPM, SDRAM, and Flash memory are in Table 1. The parameters and values are SimpleScalar's configuration [1]. We use this setup throughout the paper.

PISA uses a 64-bit instruction word (to facilitate experimentation); however, embedded processors typically use a 16-bit or 32-bit instruction. To account for PISA's large instructions, we double the data width and size of the instruction cache (including the instruction cache block size) and scratchpad. For example, to simulate a

Table 1: Simulation configuration.

SimpleScalar Configuration (XScale PXA-270 624MHz)			
fetch:ifqsize	8	tlb:lat	30
decode:width	1	res:memport	1
commit:width	2	bpred	bimod
issue:wrongpath	true	bpred:bimod	128
lsq:size	4	issue:width	2
cache:dl1	32:32:32:f	issue:inorder	true
cache:il1 (32KB)	32:64:32:f	ruu:size	4
cache:dl1lat	1	res:ialu	1
cache:il1lat	1	res:fpmult	1
tlb:itlb	1:8192:32:f	res:imult	1
tlb:dtlb	1:4096:32:f	res:fpalu	1
mem:lat	60 12	mem:width	8

64KB SPM, we set the size to 128KB. A 128KB scratchpad with 64-bit instructions is equivalent to a 64KB scratchpad with 32-bit instructions. We refer to the smaller *effective size* (i.e., a 64KB).

The dual-issue XScale PXA-270 has 32 sets, 32-byte block, and 32-way set associative data and instruction caches. We scale the capacity of the I-cache to match the size of scratchpad memory. Our configuration uses external SDRAM with a 96*ns* access latency for the initial two words and a 19*ns* access latency for each subsequent two words on an open page. These memory latencies come from Intel documents about the XScale PXA-270 [14].

The SPM has a one cycle access; in experiments with a scratchpad, there is no instruction cache. To get access times for the Flash memory, we made measurements on a Dell Axim x50v PocketPC with NOR Flash and a 8192-byte file buffer. On this device, it takes the operating system (Windows Mobile Edition 5) 1.6*ms* to initially fetch a block into the file buffer from Flash memory and 67,700*ns* per word to read from the block.

Our experiments use the programs from MiBench [10] that our experimental setup can execute. We use the large input data sets.

3.2 Performance of Small Fragment Caches

We consider three cases. The first case is a baseline. It has no scratchpad, but it does have an L1 I-cache. Programs are run without Strata. In the second case, programs are run with Strata using a 2MB F$ in external memory. The working set of all programs fits in the 2MB F$. The third case uses SPM and there is no I-cache. We vary the scratchpad size from 16KB to 64KB. The F$ can occupy the whole SPM. In scratchpad configurations, Strata's instructions are fetched without the benefit of an I-cache (i.e., its binary image is not cached). Strata's data structures are in SDRAM. The program binary is on Flash and loaded into the F$.

Figure 2 shows the impact of F$ size. The graph reports speedup normalized to executing a program with memory shadowing (i.e., without Strata). Some results do not show in the graph; these cases have no speedup and can suffer large slowdowns. We discuss the most interesting cases in the text. The first bar ("Mem-2MB") gives the speedup when the programs are run with Strata and the 2MB F$ is in SDRAM and the I-cache is 32KB.

Programs can run faster with Strata, despite overhead imposed by the translator. For example, *jpeg.decode* has 3.2 speedup with Strata. This improvement is due to incremental loading because only a small portion of the binary image is actually exercised. With Strata, only the code that is executed is loaded into the F$, which leads to fewer accesses to Flash memory. As a result, less time is spent loading the program, which can be more easily amortized. There are three programs, *basicmath*, *fft*, *gsm.encode*, where performance suffers with Strata. *basicmath* has a 44.9% performance degradation, *fft* has a 6% degradation, and *gsm.encode* has a 15.7% degradation. Because these programs are small, their binary image can be loaded quickly with memory shadowing. As a result, there is less benefit from incremental loading. Also, memory shadowing does not incur the overhead of dynamic translation.

The remaining bars ("SP-64KB", "SP-32KB", and "SP-16KB") show the speedup when the F$ is in SPM. These results are normalized to a baseline with an I-cache that has the same capacity as the SPM. The results show that with SPM, when large enough (e.g., SP-32KB), many programs (e.g., *crc, gsm.decode, qsort, sha, stringsearch,* and *susan.smoothing*) have similar performance as Mem-2MB. In this situation, the translated code's working set fits in SPM. In essence, the scratchpad serves the same role as the I-cache. In a few cases, performance is improved. For example, *adpcm.decode* goes from a 1.7 speedup with Mem-2MB to a 1.9 speedup with SP-64KB and SP-32KB. This improvement is due to the faster effective access time with SPM and illustrates the benefit of putting the fragment cache in SPM, rather than main memory.

However, when the translated code's working set does not fit in scratchpad, performance suffers. In *rijndael.encode*, the speedup decreases from 1.9 (SP-64KB) to 1.7 (SP-32KB). This degradation is more pronounced when going from SP-32KB to SP-16KB, where SP-16KB has 81.5 slowdown. *gsm.decode* has particularly dramatic behavior: Its performance goes from a 1.8 speedup (SP-64KB) to a 491.2 slowdown (SP-16KB)! This benchmark thrashes *badly* in small fragment caches.

The reason some programs do worse with a small SPM is due to the F$ eviction policy. A typical eviction strategy flushes the entire F$ when its capacity is exceeded [2]. Too many flushes lead to poor performance. Table 2 shows the number of flushes for each SPM size. The programs usually have zero or one flush for SP-64KB. When the scratchpad is 32KB or 16KB, there can be many more flushes. For example, *gsm.encode* has one flush in SP-64KB, 10,862 flushes in SP-32KB, and 26,162 flushes in SP-16KB.

These results show that flushing and subsequently filling the F$ can harm performance. There are two parts to this problem. First, the cost of fetching an untranslated instruction is large due to the Flash memory. Even with file buffers, the Flash memory has a high effective access latency. Second, the number of flushes is important, given the latency of reading instructions from Flash memory. In Section 4, we describe three techniques to address these problems. We focus on SP-32KB to keep the results presented manageable. Since our techniques address programs that are sensitive to F$ size, we consider only the programs from Table 2 where the number of flushes is greater than zero. In Section 5, we report full results on all programs and scratchpad configurations.

4. IMPROVING F$ PERFORMANCE

The way to improve the performance of a small F$ is to reduce the flushes and their cost. Reducing the number of flushes is similar to

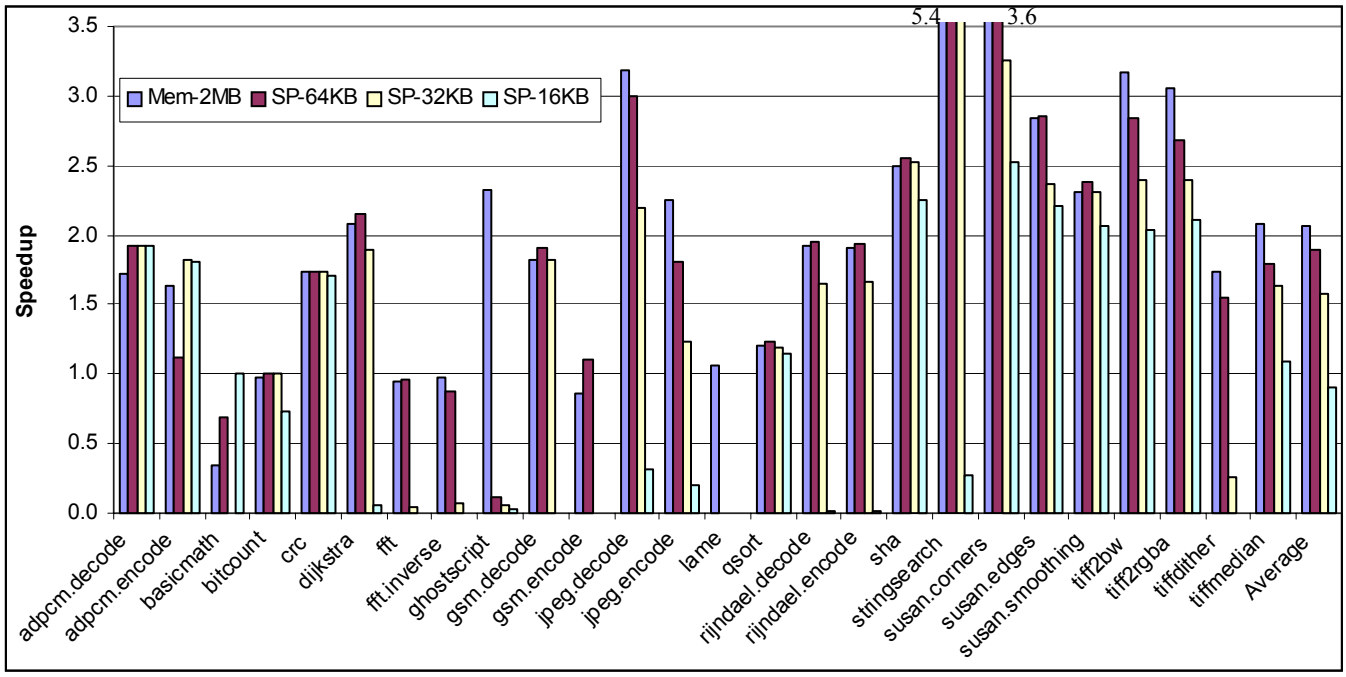

Figure 2: Speedup with a 2MB F$ in SDRAM (Mem-2MB); and F$ in 64KB (SP-64KB), 32KB (SP-32KB), 16KB (SP-16KB) SPM.

Table 2: Number of flushes for 64KB, 32KB, and 16KB F$ sizes.

Benchmark	SP-64KB	SP-32KB	SP-16KB	Benchmark	SP-64KB	SP-32KB	SP-16KB	Benchmark	SP-64KB	SP-32KB	SP-16KB
basicmath	1	25323	1587066	*tiff2bw*	1	4	10	*sha*	0	1	2
bitcount	0	0	8	*tiff2rgba*	1	4	11	*crc*	0	0	2
qsort	0	1	4	*tiffdither*	3	67	6087	*fft*	0	262	131070
susan.corners	0	1	4	*tiffmedian*	2	5	21	*fft.inverse*	1	120	113908
susan.edges	0	2	5	*dijkstra*	0	1	201	*adpcm.encode*	0	0	1
susan.smoothing	0	1	4	*ghostscript*	437	1674	9626	*adpcm.decode*	0	0	1
jpeg.encode	2	8	92	*stringsearch*	0	0	34	*gsm.encode*	1	10862	26162
jpeg.decode	1	5	71	*rijndael.encode*	0	1	867	*gsm.decode*	0	2	7856
lame	178	4494	10757	*rijndael.decode*	0	1	855				

minimizing the miss rate in traditional caches because a decrease in flushes leads to fewer overall misses to request previously translated code. Likewise, the cost of refilling the F$ and loading code from Flash is similar to the miss penalty. We use *footprint reduction* to minimize the amount of code generated by the dynamic translator so that more of the working set can be captured. With a small footprint, there will be fewer F$ flushes. Nevertheless, this reduction does not guarantee that the working set will fit in the F$. We minimize the penalty of translating previously evicted instructions by memoizing them with *victim caching*. The memoized instructions can be accessed more quickly than the untranslated ones on Flash. Lastly, *fragment pinning* avoids unnecessarily evicting and memoizing needed code. Essentially, we reduce the miss rate and the miss penalty of a small F$. In this way, the F$ can be allocated to SPM to gain its advantages.

4.1 Footprint Reduction

Footprint reduction minimizes code expansion due to control transfer instructions. A DBT system typically generates extra instructions at each application control transfer. These instructions re-enter the translator when an untranslated application address is encountered. For example, consider conditional branches. Assume that a fragment is a basic block (it can be a general code region). When a branch is translated whose taken and not-taken targets are not in the F$, the translator generates a *trampoline* for each branch direction. A not-taken trampoline is created to return control to the translator when the not-taken direction is executed. Similarly, a taken trampoline is created. Once the DBT system translates the instructions at an application address, any trampolines for that address can be rewritten (i.e., linked) to directly transfer control to the corresponding fragment. Indirect branches have similar trampolines. However, an indirect branch's target address can change during subsequent executions of the branch. As a result, indirect branches can not be linked to their target fragments because targets are unknown until the branch executes.

A trampoline instance may be generated for each translated branch. By generating the trampolines in a specific context, their

Table 3: Number of flushes without and with footprint reduction.

Benchmark	No FR	NE	NE+TF	Benchmark	No FR	NE	NE+TF	Benchmark	No FR	NE	NE+TF
basicmath	25323	4569	1	*jpeg.decode*	5	4	2	*susan.corners*	1	1	0
dijkstra	1	1	0	*jpeg.encode*	8	6	2	*susan.edges*	2	2	1
fft	262	12	1	*lame*	4495	3482	2324	*susan.smoothing*	1	1	0
fft.inverse	120	35	1	*qsort*	1	1	0	*tiff2bw*	4	4	1
ghostscript	1674	1514	489	*rijndael.decode*	1	1	0	*tiff2rgba*	4	3	1
gsm.decode	2	1	1	*rijndael.encode*	1	1	0	*tiffdither*	67	5	3
gsm.encode	10862	10862	88	*sha*	1	1	0	*tiffmedian*	5	4	2

Table 4: Speedup/slowdown without and with footprint reduction.

Benchmark	No FR	NE	NE+TF	Benchmark	No FR	NE	NE+TF	Benchmark	No FR	NE	NE+TF
basicmath	(554.3)	(87.3)	(1.5)	*jpeg.decode*	2.2	2.3	3.1	*susan.corners*	3.3	3.4	3.9
dijkstra	1.9	2.0	2.3	*jpeg.encode*	1.2	1.6	2.1	*susan.edges*	2.4	2.5	2.9
fft	(27.6)	(1.9)	(1.03)	*lame*	(259.9)	(187.5)	(185.6)	*susan.smoothing*	2.3	2.4	2.5
fft.inverse	(15.4)	(4.5)	(1.1)	*qsort*	1.2	1.2	1.2	*tiff2bw*	2.4	2.7	3.3
ghostscript	(16.3)	(14.3)	(8.4)	*rijndael.decode*	1.6	1.7	2.0	*tiff2rgba*	2.4	2.9	3.1
gsm.decode	1.8	1.9	1.9	*rijndael.encode*	1.7	1.7	2.0	*tiffdither*	(4.0)	1.7	1.7
gsm.encode	(938.3)	(928.3)	(11.2)	*sha*	2.5	2.6	2.6	*tiffmedian*	1.7	1.9	2.0

instruction count (IC) can be minimized. Reducing the IC is important for an indirect branch because its trampoline will not be removed and may be executed many times [12]. While this approach minimizes IC, it causes code expansion. Indeed, once a fragment is linked to another fragment, the F$ space occupied by the trampoline can not be easily reclaimed. For example, Strata's not-taken and taken trampolines each take seven PISA instructions. When a trampoline is replaced by a link to a target fragment, the link needs one instruction. The remaining six instructions from the original trampoline are unused holes in the F$. They are too small to hold a new fragment (including trampolines).

To minimize footprint, we trade a trampoline's IC for space. We use a *trampoline function* so all conditional branches can share one trampoline. A call is made to the function from the site of a conditional branch. Although the trampoline function increases IC, the call site needs only two instructions. The F$ space associated with the call is better utilized when a fragment is linked, avoiding the wasted space associated with per-branch trampolines.

Indirect branches are handled similarly with an *indirect trampoline function*. The indirect trampoline function turns Strata's shared indirect branch translation cache lookup into a function [12]. The indirect trampoline function is allocated to the scratchpad to make it as fast as possible. This function maps an indirect branch's target address to a fragment. If a target address is in the F$, then control is directly transferred to the fragment. If the target is not in the F$, then the DBT system is re-entered to translate the missing code.

The policy used by the dynamic translator for handling unconditional branches and calls is also important to code expansion. One common policy for unconditional branches and calls is to eliminate them during code translation [13]. Unconditional branch elimination replaces an unconditional branch with the basic block at its target address. Similarly, partial call inlining inserts the first basic block from a called function. While these techniques reduce IC, they cause code duplication and the increase in the code footprint may not warrant the IC reduction in a small SPM. To avoid this situation, we simply disable all techniques that duplicate instructions.

4.1.1 Experimental Results

Using the setup from Section 3.1, we investigated how footprint reduction improves performance. We report two results: The reduction in F$ flushes and program speedup/slowdown. The number of F$ flushes is related to code expansion; with less code expansion, there will be fewer flushes. The speedup is improved by avoiding flushes. We examined three Strata configurations. The first configuration, "No FR" (No Footprint Reduction), is the standard one used by Strata [13]. Configuration "NE" (Never Elide) changes Strata's standard configuration to disable eliding of unconditional jumps and partial inlining of calls. "NE+TF" (Never Elide and Trampoline Functions) is the NE configuration with trampoline functions for direct and indirect branches.

Table 3 shows the number of F$ flushes without and with footprint reduction. The columns labeled "No FR" are the values from Table 2. Footprint reduction can reduce the flushes. For example, in *basicmath*, NE decreases the flushes by a factor of five. With NE+TF, *basicmath* has only one flush. The reason NE and NE+TF are so effective in some programs is due to significant code expansion. For example, *basicmath* is a series of loops that do not initially fit in the F$. With NE+TF, the loops fully fit. *fft* and *fft.inverse* are similar. Although *lame*, *ghostscript*, and *gsm.encode* have a decrease in flushes, they still experience a relatively high flush rate. For example, *ghostscript* has 1,674 flushes without footprint reduction and 489 with NE+TF.

With a decrease in the number of flushes, there is a performance improvement, as shown in Table 4. Values without parentheses are speedups and values with parentheses are slowdowns. For example, *dijkstra* improves from a speedup of 1.9 without footprint reduction to 2.3 with NE+TF. After applying NE+TF, there are no F$ flushes. *tiffdither* is also an interesting case, where there is initially a slowdown. The number of flushes goes from 67 without footprint reduction to 3 with NE+TF. Without footprint reduction, it has a slowdown of 4 and with NE+TF, it has a speedup of 1.7. A final example is *fft*, where the initial slowdown is 27.6 and when

NE+TF is used, it is only a 1.03. We conclude that footprint reduction is effective and can lead to a performance gain.

Despite the benefit of NE+TF, some programs (e.g., *lame*, *ghostscript*, and *gsm.encode)* still have many flushes and high slowdowns. To address this situation, we use victim compression and pinning. Since these techniques are only beneficial when there is at least one flush, we further consider the programs from Table 3 with one or more flushes under NE+TF.

4.2 Victim Compression

When the fragment cache becomes full, it is flushed when a new fragment is created. Although more sophisticated eviction policies are possible, flushing the F$ is a simple one that is used to minimize the overhead of managing the F$ [5,11,27]. The problem with this approach is that fragments which are soon needed can be evicted, especially for a small F$. In the case of Flash storage, a high access cost will be paid to re-translate these fragments.

To address this problem, we introduce *victim compression*, which is inspired by a hardware victim cache [15]. Our approach saves the victims that are evicted when the fragment cache is flushed. Because there are as many victims as fragments, we compress the evicted code. Code compression minimizes the amount of space needed for the victims. The compressor/decompressor is implemented in software (part of Strata), without additional hardware.

A compressed version of the evicted code is kept in a region of the F$, called the *compressed fragment region (CFR)*. Whenever the translator fetches an application address, it consults the CFR to see whether a corresponding fragment has been translated. If so, the fragment is fetched from the CFR and decompressed. The code fetched from Flash or decompressed from the CFR is put into a F$ region called the *executable fragment region (EFR)*. As long as the cost of decompression is less than the cost of accessing Flash and retranslating the fragment, then victim compression will reduce the penalty of populating the F$ with previously seen fragments.

There are many design choices for victim compression. We examine two important ones: 1) how to partition and manage the F$ among compressed and uncompressed fragments and 2) how to include compression and decompression in the translator. We note that a third design choice is where to place the compressed code; it could be put in external memory. Our aim is to keep the translated code footprint, including uncompressed and compressed code, minimal. Thus, we allocate the compressed code to the F$ in SPM, which ensures that the total space needed to store the translated code is not increased beyond the original F$ size.

4.2.1 Victim Compression Strategy

The first design question concerns how to partition the fragment cache into compressed and uncompressed code. One possible scheme is to statically partition the fragment cache's space into executable (EFR) and compressed code (CFR). However, this scheme reduces the effective size of the F$ because only a fixed portion of the F$ can hold executable, translated code. As a result, it will lead to many more flushes and poor program performance.

Rather than statically allocating a fixed amount of memory to the EFR and CFR, we use a *variable region partitioning* scheme.This scheme addresses the limitations of the fixed scheme because the EFR and CFR dynamically change size. The translator fills the EFR from Flash memory or the compressed fragment region. The CFR starts at a high address, while the executable fragment region starts at a low address. If the upper boundary of the EFR reaches the lower boundary of the CFR, then the compressed fragments are purged. The space occupied by the compressed code is freed and can be used to hold executable code. If the EFR exceeds the capacity of the whole fragment cache, it is flushed. In this case, the EFR is compressed and a new CFR is allocated to hold the victims.

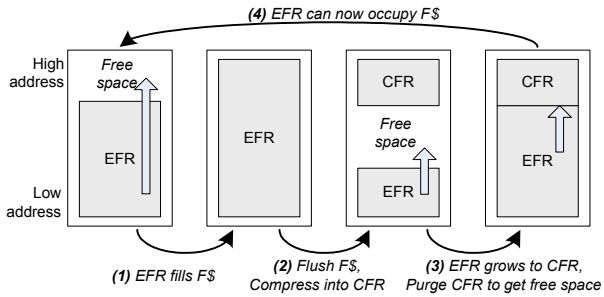

Figure 3: Steps for variable region partitioning.

Figure 3 illustrates the steps for variable region partitioning. In the first step, there is no compressed code and the executable code may occupy the whole fragment cache. In the second step, when the EFR overflows, it is flushed. The evicted fragments are compressed. In the third step, the CFR and Flash memory are used to retrieve previous fragments and application code. In the fourth step, when the EFR reaches the CFR, the compressed fragments are purged. This space can be used to hold executable code. Any new application addresses fetched by the translator will come from Flash. These steps will repeat throughout program execution.

The advantage to this scheme is that the EFR can occupy the entire fragment cache when necessary, which may lead to fewer flushes. Additionally, the CFR size can adapt to the compression ratio. However, the disadvantage is that the CFR captures only the *most recent victims* and does not persist across fragment cache flushes.

4.2.2 Incorporating Compression/Decompression

The second design question is how to incorporate the code compressor and decompressor into the dynamic translator. Figure 4 shows how we integrated compression and decompression into Strata. Other dynamic translators operate similarly and compression/decompression can be incorporated in their fetch-translate-execute loop. The shaded region in the upper right corner of the figure is the compression loop. The loop is entered when the F$ is flushed. The decompression loop is in the lower right corner. It is entered when an application address is help as compressed code.

The compression loop is entered when the EFR is flushed. On a flush, the F$ is traversed to find code that was inserted by the dynamic translator. These instructions are not compressed since they can be regenerated when the fragment is decompressed. In fact, branch addresses in these instructions depend on F$ layout. The layout of fragments will change after a flush since it depends on the order in which application addresses are requested. Next, two dictionaries are constructed: one dictionary (`csym`), is used for compression and the other (`dsym`) is used for decompression.

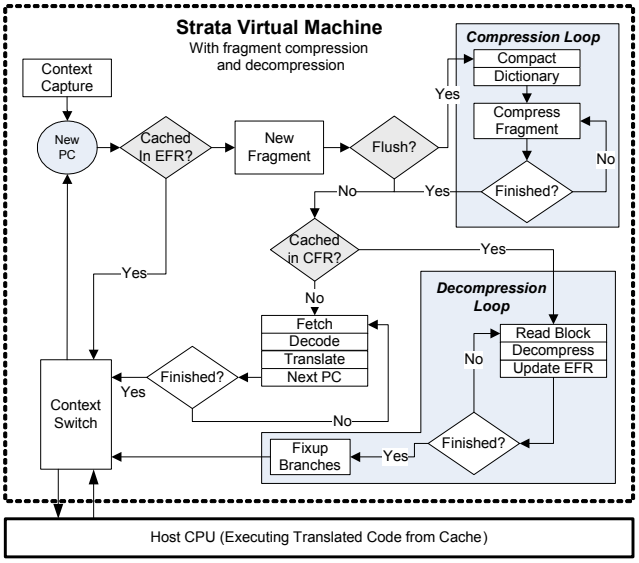

Figure 4: Incorporating compression and decompression.

Once the dictionaries are constructed, the EFR is compressed. For decompression, it is necessary to know which application addresses are in the CFR. The compressor builds a fragment map (`cfMap`) that relates application addresses to their corresponding compressed fragments. `csym` is discarded after compression. `cfMap` and `dsym` are persistent and stored in external SDRAM as data. They are accessed during decompression.

The decompression loop is entered when the translator fetches a new application address. On a fetch, the translator checks whether that address is held in the CFR. A lookup is done in `cfMap` to see whether there is a compressed fragment for the application address. If so, the fragment is decompressed and the control code needed by the translator is generated.

We use a compression algorithm that is based on IBM's CodePack algorithm [16]. CodePack has a good compression ratio (about 50%) and performance when implemented in software [23]. We changed CodePack to accommodate PISA and to avoid aligning the starting point of compressed code to word boundaries (which minimizes the footprint of the compressed code in the CFR). Our implementation is configurable to support other compression algorithms. The inclusion of other algorithms does not change the process outlined in Figure 4.

4.2.3 Experimental Results

With variable region partitioning, the number of flushes is the same as NE+TF. Because this scheme discards the compressed code, it can adapt to situations when the EFR needs more capacity. In fact, the executable fragment region effectively has the same F$ capacity as NE+TF. The benefit to victim compression will show as an improvement in performance by avoiding accesses to Flash.

Table 5 shows performance without (NE+TF) and with victim compression (Comp.). When a program suffers at least one F$ flush, there is potential improvement by memoizing the victims. For example, *fft* and *jpeg.decode* have the same number of flushes with NE+TF and victim compression. Their performance is improved since the accesses to Flash are avoided when evicted code is needed again. This result also shows that there is code reuse across EFR flushes. Indeed, when a program has many flushes and there is much reuse, victim compression is especially beneficial. For example, *ghostscript* has a 8.4 slowdown with NE+TF and a 4.5 slowdown with victim compression. In this program, some code is evicted for a short period and reused later, possibly after more than one flush has occurred. Another interesting case is *fft.inverse*, where an initial 1.1 slowdown is improved to a 1.1 speedup.

Table 5: Speedup/slowdown with victim compression.

Benchmark	NE+TF	Comp.	Benchmark	NE+TF	Comp.
basicmath	(1.5)	(1.4)	*gsm.encode*	(11.2)	(3.3)
fft	(1.03)	1.0	*jpeg.decode*	3.1	3.2
fft.inverse	(1.1)	1.1	*jpeg.encode*	2.1	2.5
ghostscript	(8.4)	(4.5)	*lame*	(185.6)	(158.8)
gsm.decode	1.9	1.9	*susan.edges*	2.9	3.0
tiff2bw	(3.3)	(3.7)	*tiffdither*	1.7	1.9
tiff2rgba	3.1	3.5	*tiffmedian*	2.0	2.3

`cfMap`, `csym` and `dsym` are added for victim compression, which increases main memory usage by Strata. Without victim compression, memory usage for Strata's data structures ranges from 118.7 KB (*gsm.encode*) to 148.2 KB (*tiffdither*), with an average 137 KB. In comparison, victim compression has a near constant 90% increase in memory usage. This increase is primarily due to `cfMap`, which tracks information about the compressed fragments. Finally, with compression, Strata's binary image is increased by 16 KB. We conclude that the performance gain from victim compression is worth its small increase in SDRAM memory usage.

Because fragment pinning, which is described next, benefits programs with more than one fragment cache flush, we now further consider only such programs.

4.3 Fragment Pinning

While victim compression avoids fetching and translating previously encountered code, it still suffers overhead. For a small F$, it is possible that the same fragment may be evicted, compressed, and decompressed many times, incurring unnecessary overhead. To address this problem, we use *fragment pinning*, where a fragment can be locked in the F$. When a fragment is pinned, it is not evicted on a flush. It remains as executable code and will not incur multiple compression and decompression cycles.

We incorporate pinning with a new fragment cache region, called the *pinned fragment region (PFR)*. The code in the PFR is executable, but persists across F$ flushes. The PFR and EFR are intermixed to best utilize fragment cache space. During runtime, fragments are moved between the EFR, CFR, and the PFR according to a *pinning strategy*. The pinning strategy decides what fragments to pin, when to pin them, and when to release the pins.

4.3.1 Pinning Strategy

There are many possible pinning strategies. One strategy might count fragment execution frequency to pin hot fragments. Counters could also be used to determine when to release a cold pinned frag-

ment. However, these strategies have monitoring overhead, whether done with instrumentation or hardware counters.

Instead, we take advantage of the fact that the compressed fragment region holds recent victims. If an application address is needed and a corresponding fragment is in the CFR, then that fragment is likely part of the current working set. When a needed fragment is in the CFR, it is decompressed, pinned, and put into the pinned fragment region. Thus, the fragments that can be pinned are victims from a previous F$ flush. A pin is acquired immediately once an application address is requested and its fragment is found in the CFR. The pins are released when the size of the PFR reaches a "release threshold" ratio of the executable fragment region. Pins are released when $\frac{sizeof(PFR)}{sizeof(PFR)+sizeof(EFR)} \geq threshold$. The intuition is that a working set change is most likely when the executable fragment region puts pressure on F$ capacity. The pins are released so that the fragments do not become stale. This strategy is simple and inexpensive because it does not need monitoring.

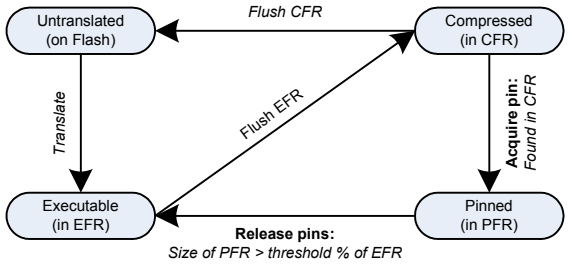

Figure 5: Pinning state diagram

Figure 5 illustrates our pinning strategy as a state machine. The diagram shows the states that a fragment goes through and the transitions that cause a state change. Initially, an application address is in the untranslated state. When the address is fetched, a fragment is created and put in the EFR. If the EFR is flushed, the fragment is put in a compressed state. When the address is requested again, the corresponding fragment is transitioned from a compressed state to a pinned state. The pin will be released when the PFR is above the threshold ratio. A CFR flush causes all compressed code to be transitioned to untranslated states.

4.3.2 Experimental Results

We investigated how fragment pinning helps reduce F$ flushes. Table 6 shows the number of flushes without and with fragment pinning. In the table, "Comp." is victim compression with variable region partitioning and "Pin" is compression with pinning. "PinFIFO" is compression with a variant on the pinning scheme. In PinFIFO, pins are released in FIFO order. In this policy, when the EFR causes a flush, pins are successively released until the pinned code is under the release threshold. Thus, pins are released in a more fine-grained fashion to avoid releasing them too early. In both Pin and PinFIFO, the pin release threshold is 50%. We tried several thresholds and 50% did the best on average.

Fragment pinning can reduce flushes. For example, in *ghostscript*, the number of flushes changes from 489 with compression to 196 with Pin. When FIFO information is used (PinFIFO), there is only one flush. After the EFR is flushed one time, there will *always* be at least one pinned fragment in the EFR with this strategy. Thus, the *whole* F$ is never flushed again. The EFR may continue to overflow its boundaries, causing the CFR to be discarded.

Table 6: Number of flushes without and with pinning.

Benchmark	Comp.	Pin	PinFIFO
ghostscript	489	196	1
gsm.encode	88	16	1
jpeg.decode	2	1	1
jpeg.encode	2	1	1
lame	2320	235	1
tiffdither	3	2	1
tiffmedian	2	1	1

Table 7: Speedup/slowdown with pinning.

Benchmark	Comp.	Pin	PinFIFO
ghostscript	(4.5)	(5.6)	(3.6)
gsm.encode	(3.3)	(3.1)	(1.1)
jpeg.decode	3.2	3.2	3.2
jpeg.encode	2.5	2.5	2.5
lame	(158.8)	(143.1)	(114.3)
tiffdither	1.9	1.9	2.0
tiffmedian	2.3	2.3	2.3

Table 7 gives performance with pinning. For *ghostscript*, there is a 3.6 slowdown with PinFIFO and 4.5 with compression. Pin, on the other hand, increases the slowdown for *ghostscript* because older, unneeded fragments are kept pinned, which is avoided by PinFIFO. *gsm.encode* also has a particularly good improvement with pinning: It has a 3.3 slowdown with compression and only a 1.1 slowdown with PinFIFO. The slowdown for this benchmark is improved enough to be competitive with traditional memory shadowing. From these results, we conclude that pinning is beneficial when evicted code may be requested many times.

5. OVERALL IMPROVEMENT

Figure 6 shows the performance improvement for all benchmarks and SPM sizes when footprint reduction, victim compression and pinning (PinFIFO, 50% threshold) are enabled. The figure shows performance from Figure 2 for comparison.

Our techniques improve performance across the SPM sizes, particularly when the translated code working set does not initially fit in the F$. For example, in *dijkstra* for SP-16KB, performance is improved from a 15.8 slowdown to a 2.2 speedup. *gsm.encode* has an impressive improvement for SP-32KB. It initially has a slowdown of 938.3 due to thrashing; with out techniques, its slowdown is reduced to 1.1 because it no longer thrashes. These results also show that our techniques usually do not degrade performance when unneeded. For example, *adpcm.decode* is a tight loop that fits in all SPM sizes and its speedup is 1.9 in all cases.

Even with our techniques, some programs still have large runtime overheads. *lame* has this behavior. With SP-32KB, it has an initial slowdown of 258.7 and a final slowdown of 114.3. *lame*'s working set does not fully fit in the F$. Although PinFIFO reduces flushes to one, the executable fragment region overflows and there are many accesses to Flash memory. *ghostscript* behaves similarly, but the effect is not as dramatic.

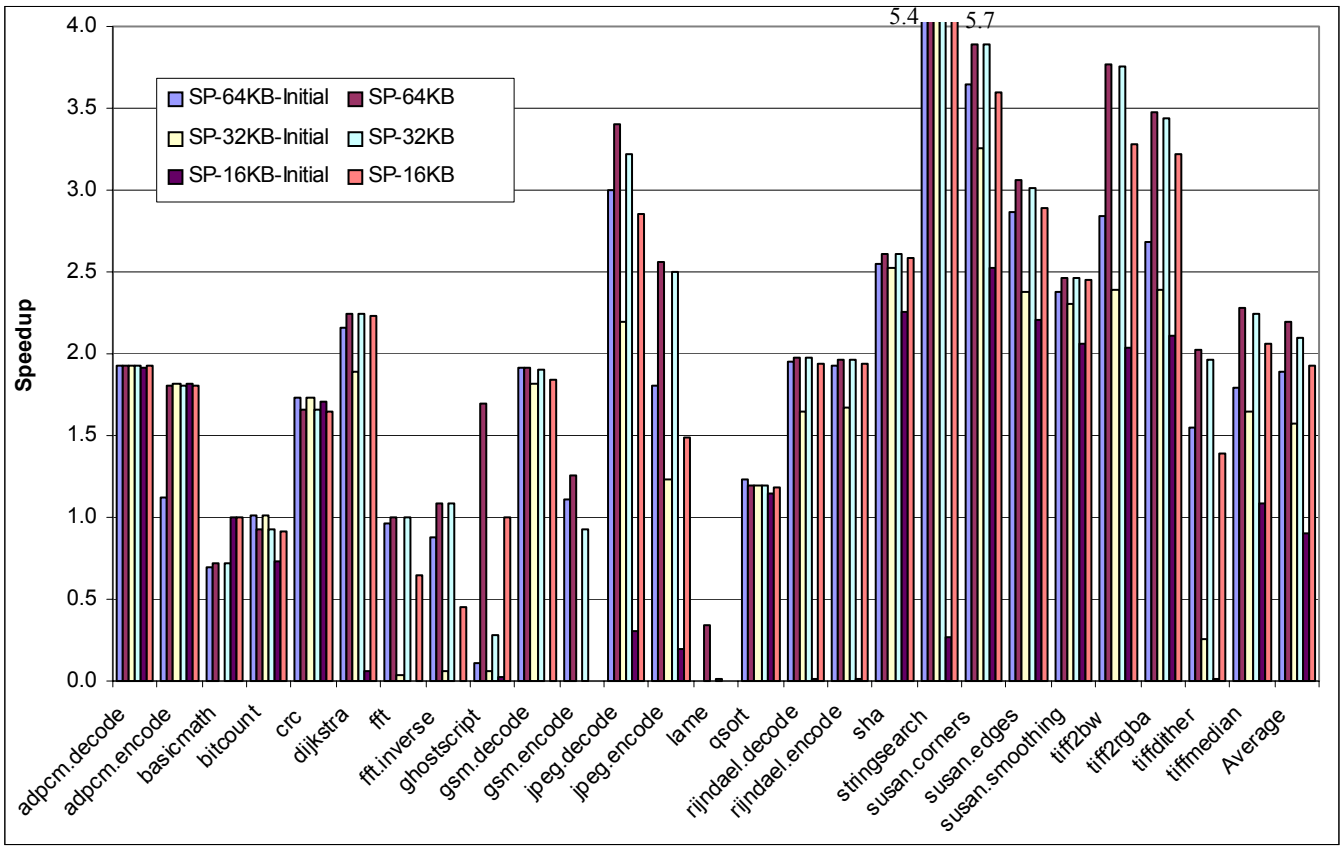

Figure 6: Initial speedup compared to final speedup. SP-64KB-Initial, SP-32KB-Initial and SP-16KB-Initial are the initial speedups.

On average, footprint reduction, victim compression and pinning, have an initial speedup of 1.9 (SP-64KB), 1.6 (SP-32KB), and 0.9 (SP-16KB) over memory shadowing. With our techniques, these average speedups are improved to 2.2 (SP-64KB), 2.1 (SP-32KB) and 1.9 (SP-16KB).

In comparison to Mem-2MB, a 32KB fragment cache allocated to SPM (i.e., SP-32KB) has slightly better performance. Mem-2MB has an average speedup of 2.06 and SP-32KB has an average speedup of 2.1. The total amount of memory needed for the fragment cache is much less with SP-32KB than Mem-2MB, yet its performance is better than Mem-2MB. From these results, we conclude that our techniques are effective and will help enable the use of DBT in embedded systems with small scratchpads.

6. RELATED WORK

There has been much work related to this paper. The management of the F$ has been widely studied for general-purpose systems and large fragment caches [2, 3, 4, 5, 11, 21, 23, 27]. Bala et al. suggested the strategy of flushing the F$ when it becomes full [2]. Hazelwood and Smith investigated a generational garbage collection approach that promotes traces when they are frequently used [11]. Bruening et al. investigated how to manage the F$ for multi-threaded programs [5]. They also investigated how to bound the F$ and maintain consistency with self modifying code [4].

Zhou et al. investigated F$ management for smartcards [27]. This work used a code server to download on-demand code into the F$. Their approach used a profile-based technique to pre-plan cache decisions so they could be done with low cost. Instead, our approach makes all decisions online. Guha, Hazelwood and Soffa suggested a technique to reduce code footprint due to exit stubs in a program instrumenter [9]. Other related work comes from Shogan and Childers [23]. This project proposed compressing the program binary prior to execution. The image is decompressed by the dynamic translator. None of the approaches used SPM.

The most related work addresses software-managed instruction caches [19] for processors that lack an instruction cache. This approach minimizes the cost of loading and evicting instructions from the scratchpad. It is similar to incremental loading done by a dynamic translator. They used static binary rewriting to manage the scratchpad. Pinning was used to keep important basic blocks in the scratchpad. However, the project did not apply their techniques in a dynamic translator or use code compression.

Lastly, there has been much work on managing code in SPM with the compiler. Approaches proposed for code execution in the scratchpad use compiler support to partition the code and allocate it statically to the scratchpad [7, 8, 25], or copy it on-demand from main memory to the scratchpad [7, 8, 20]. Code is selected for SPM execution at compile time [25], or based on profiles [7, 8, 20], to optimize performance or energy consumption.

7. CONCLUSION

This paper investigated how to use scratchpad memory for dynamic binary translators. We described three techniques to improve the performance from a small F$ when allocated to SPM. Footprint reduction mitigates code expansion, while victim compression reduces the cost associated with fetching and translating previously encountered code. The last technique, fragment pinning, locks important code in the F$. Our techniques improved performance of small fragment caches by an average of 20% in a 64KB SPM, 30% in a 32KB SPM and 110% in a 16KB SPM. Indeed, with our techniques, a much smaller 32KB F$ allocated to the scratchpad has equivalent performance as a 2MB fragment cache allocated to main memory.

8. ACKNOWLEDGEMENTS

This research was supported by the National Science Foundation under grants CNS-0551492, CNS-0509115, CNS-0305198, CNS-0305144, CNS-0524432, and CNS-0551560.

9. REFERENCES

[1] T. Austin, E. Larson and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling", *IEEE Computer*, Vol. 35, No. 2, February 2002.

[2] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system", *Programming Language Design and Implementation*, 2000.

[3] D. Bruening, T. Garnett and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization", *Int'l. Symp. on Code Generation and Optimization*, 2003.

[4] D. Bruening and S. Amarasinghe, "Maintaining Consistency and Boundary Capacity of Software Code Caches", *Int'l. Symp. on Code Generation and Optimization*, 2005.

[5] D. Bruening, V. Kiriansky, T. Garneet and S. Banerji, "Thread-Shared Software Code Caches", *Int'l. Symp. on Code Generation and Optimization*, 2006.

[6] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher, "DELI: A new run-time control point", *Int'l.Symp. on Microarchitecture*, 2002.

[7] B. Egger, C. Kin, C. Jang, Y. Nam, J. Lee and S. L. Min, "A Dynamic Code Placement Technique for Scratchpad Memory Using Postpass Optimization", *Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, 2006.

[8] B. Egger, J. Lee and H. Shin, "Scratchpad Memory Management for Portable Systems with a Memory Management Unit", *Conf. Embedded Software*, 2006

[9] A. Guha, K. Hazelwood and M. L. Soffa, "Reducing Exit Stub Memory Consumption in Code Caches", *High Performance Embedded Architectures and Compilers*, 2007.

[10] 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", *IEEE 4th Workshop on Workload Characterization*, 2001.

[11] K. Hazelwood and M. D. Smith, "Managing Bounded Code Caches in Dynamic Binary Optimization Systems", *ACM Trans. on Architecture and Code Optimization*, Vol. 3, No. 3, September 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", *Int'l. Symp. on Code Generation and Optimization,* 2007.

[13] J. D. Hiser, D. Williams, A. Filipi, J. W. Davidson and B. R. Childers, "Evaluating Fragment Creation Policies for SDT Systems", *Virtual Execution Environments Conf.*, 2006.

[14] Intel, *Intel PXA27x Processor Family*, document number 280004-002, August 2004.

[15] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers", *Int'l. Symp. on Computer Architecture*, 1990.

[16] T. Kemp, R. Montoye, J. Harper, J. Palmer, and D. Auerbach, "A decompression core for PowerPC", *IBM Journal of Research and Development*, 42(6), Nov. 1998.

[17] V. Kiriansky, D. Bruening and S. Amarasinghe, "Secure execution via program shepherding", *USENIX Security*, 2002.

[18] C. Luk, R. Cohn, R. Muth, et. al, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation", *Conf. on Programming Design and Implementation*, 2005.

[19] J. E. Miller and A. Agarwal, "Software-based Instruction Caching for Embedded Processors", *Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006.

[20] N. Nguyen, A. Dominguez and R. Barua, "Memory Allocation for Embedded Systems with a Compile-Time-Unknown Scratch-Pad Size", *Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, 2005

[21] K. Scott, N. Kumar, S. Veluswamy, B. Childers, J. Davidson, M. L. Soffa, "Reconfigurable and retargetable software dynamic translation", *Symp. on Code Generation and Optimization*, 2003.

[22] K. Scott and J. Davidson, "Safe virtual execution using software dynamic translation", *Annual Computer Security Applications Conference*, 2002.

[23] S. Shogan and B. R. Childers. "Compact Binaries with Code Compression in a Software Dynamic Translator," *Conf. Design Automation and Test in Europe*, 2004.

[24] S. Sridhar, J. S. Shapiro, E. Northup, P. P. Bungale, "HDTrans: An open source, low-level dynamic instrumentation system", *Virtual Execution Environments Conf.*, 2006.

[25] S. Steinke, L.Wehmeyer, B. Lee and P. Marwedel, "Assigning Program and Data Objects to Scratchpad for Energy Reduction", *Conf. Design, Automation and Test in Europe*, 2002

[26] Q. Wu, M. Martonosi, D. W. Clark, Y. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks, "A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance", *Int'l. Symp. on Microarchitecture,* 2005.

[27] S. Zhou, B. R. Childers, and M. L. Soffa. Planning for Code Buffer Management in Distributed Virtual Execution Environments, *Virtual Execution Environments Conf.*, 2005.