Addressing the Challenges of DBT for the ARM Architecture

Ryan W. Moore José A. Baiocchi
Bruce R. Childers
University of Pittsburgh
{rmoore,baiocchi,childers}@cs.pitt.edu

Jack W. Davidson Jason D. Hiser
University of Virginia
{jwd,hiser}@virginia.edu

Abstract

Dynamic binary translation (DBT) can provide security, virtualization, resource management and other desirable services to embedded systems. Although DBT has many benefits, its run-time performance overhead can be relatively high. The run-time overhead is important in embedded systems due to their slow processor clock speeds, simple microarchitectures, and small caches. This paper addresses how to implement efficient DBT for ARM-based embedded systems, taking into account instruction set and cache/TLB nuances. We develop several techniques that reduce DBT overhead for the ARM. Our techniques focus on cache and TLB behavior. We tested the techniques on an ARM-based embedded device and found that DBT overhead was reduced by 54% in comparison to a general-purpose DBT configuration that is known to perform well, thus further enabling DBT for a wide range of purposes.

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

General Terms Measurement, Performance, Design, Experimentation

Keywords ARM, Dynamic Binary Translation, Virtualization

1. Introduction

Dynamic binary translation (DBT) is a technology that allows monitoring and possibly modifying a software application as it runs. With information available at run-time, DBT can be used in embedded systems for platform emulation [5], code security [7], program analysis and instrumentation [9], power management [20], virtualization [1], dynamic optimization [4], and other purposes. For example, in embedded systems with Flash memory, DBT can be used to provide binary loading [3]. An incremental loader built with DBT can achieve a 1.9 to 2.2 speedup over traditional memory shadowing.

During translation, application code is modified and saved into a software-managed buffer, called the *fragment cache* (F$), from which the code executes. The translated code invokes the dynamic translator whenever a new application address is encountered. Naturally, dynamic translation introduces run-time performance overhead. The overhead comes from the translation process and the code injected by the translator for its own purposes (e.g., to remain in control of the application). This run-time overhead, if too large, can dissuade users from taking advantage of DBT because it might require a faster CPU. Recent work with scratchpad memory has shown that DBT can achieve low run-time overhead [2, 3]. This paper focuses on ARM-based embedded systems without scratchpad as a step toward building an efficient dynamic translation infrastructure for ARM.

The ARM architecture is dominant in embedded systems. It is used in numerous devices, such as cell phones, routers, automobile control systems, HDTV set-top boxes, and personal digital assistants. To make DBT practical for the ARM, and thus, widely applicable, it has to be carefully tuned, especially because the ARM exposes the program counter (PC) in a register (`r15`).

The exposed PC is common in ARM code. For example, it is used in load instructions to make PC-relative references to large constants (due to ARM's small 8-bit immediate field[1]). Writes to the exposed PC are also common and used to implement indirect calls, returns, switch statements, etc. The way the dynamic translator handles the frequent reads and writes of the exposed PC has a significant run-time overhead impact. References to the PC are relative to untranslated application code, rather than translated code. Copying an instruction into the F$ changes its PC. If the original instruction referenced the PC, the translator must map the new PC reference of the moved instruction to the original one. Exposed PC values that are available during translation can be mapped with emulation. PC values that are unknown during translation require dynamic mapping. Indirect branches, for example, jump to a computed target address. A commonly used, fast mechanism to map an indirect branch target address is the *Indirect Branch Target Cache* (IBTC) [10].

Exposed PC emulation and the IBTC have to be carefully configured to have low run-time DBT overhead on the ARM. The IBTC, in particular, has many configuration options [10]. In this paper, we examine an IBTC configuration that past work recommends for general-purpose systems [10] and determine it is not well suited for the ARM's ISA idiosyncrasies. We tune the IBTC to find its best configuration for the Intel/Marvell PXA270 ARM processor. We also develop a new low overhead mechanism that is specialized to the handling of returns. Our implementation is finely tuned, with hand-crafted assembly routines to perform context switches and other low-level translation tasks.

Despite much tuning, there is room for improvement in the translator's run-time overhead. To better understand the overhead sources, we profiled the programs with and without DBT using hardware performance counters. The performance counters showed

[1] The 8-bit immediate can optionally be shifted before use.

©ACM 2009. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version will be published in the Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, June 19–20, 2009.

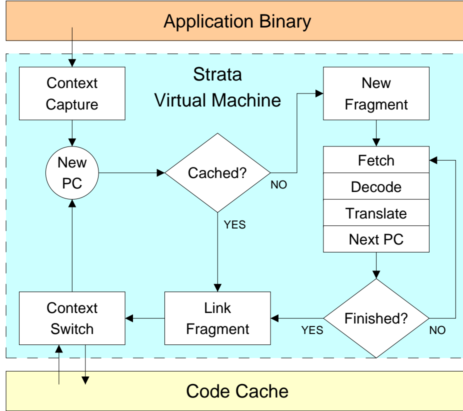


Figure 1. Dynamic binary translator system operation

Instruction	Emulation

`Addr: add r0,pc,#const`	`TAddr: ldr r0,[pc,#(Value-TAddr-8)]`
	`...`
	`Value: Addr+8+#const`

`Addr: sub r0,pc,r1`	`TAddr: ldr r0,[pc,#(Value-TAddr-8)]`
	`sub r0,r0,r1`
	`...`
	`Value: Addr+8`

`Addr: and r0,pc,r0`	`TAddr: str r1,[pc,#(Spill-TAddr-8)]`
	`ldr r1,[pc,#(Value-TAddr-12)]`
	`and r0,r1,r0`
	`ldr r1,[pc,#(Spill-TAddr-20)]`
	`...`
	`Value: Addr+8`
	`Spill:` *value of* `r1`

Figure 2. Emulation of instructions that read the exposed PC

that DBT was causing significant additional pressure on the TLBs and caches, resulting in poor performance. We identified a dominant source of the pressure as PC-relative loads and stores of values generated by the translator and from the original application code. The issue is not in the translator's generated code, but rather the *interaction* with TLBs and caches.

To improve the handling of PC-relative memory operations, we developed several new techniques, including *fragment pooling* (FP), *page pooling* (PP), *page pooling on code pages* (PPC), and *paged translator re-entry* (PR). FP, PP, and PPC lower the overhead of managing data values in the translated code. These techniques reduce overhead by grouping data in a way that improves cache and TLB behavior. PR reduces the overhead due to constants needed to re-enter the translator from the translated code. PR is important during program phase changes, when the translator may be entered frequently. PR improves the data TLB behavior due to interactions between the translated application code and the translator. Therefore, we expect our techniques to provide benefits on any ARM-based device, although the exact benefit will depend on application behavior (e.g., data usage and instruction flow) and microarchitectural features (e.g., cache and TLB contention, cost of cache and TLB misses, etc). This paper makes several contributions:

- How to tune a dynamic translator to the ARM instruction set architecture and a specific microarchitecture implementation;
- Descriptions of methods for emulation of the ARM's exposed PC, approaches for improving the IBTC on the ARM with PC-relative table accesses, and a new method to improve handling of function returns (fast checked returns);
- A comprehensive experimental evaluation of the source of runtime overhead in a carefully tuned dynamic translator for the ARM and identification of the remaining sources of overhead arising from increased cache and TLB pressure;
- Algorithm descriptions and implementations of novel techniques – fragment pooling, page pooling, page pooling with code pages, and paged translator re-entry – to reduce cache and TLB misses from data values and PC-relative constants;
- Thorough experimental evaluation of the techniques on an actual embedded device.

This paper is organized as follows. Section 2 describes the operation of DBT and how to tune a translator to the ARM. Section 3 describes new techniques to better manage constants and data values. Section 4 gives an overall evaluation. Section 5 presents related work and Section 6 concludes the paper.

2. Enabling DBT on the ARM

To enable dynamic binary translation for the ARM, attention has to be paid to several instruction set and microarchitecture features. In this section, we describe the basic operation of DBT and how to tune it for the ARM. Based on this tuned configuration, we examine the causes of program slowdown from DBT.

2.1 Dynamic Binary Translation

A high-level view of a dynamic binary translator's operation is illustrated in Figure 1. The dynamic translator must ensure that untranslated application instructions are examined and possibly modified prior to their execution. To do so, the translator is invoked whenever new application code is requested. New code is requested when execution begins and every time a control transfer instruction (CTI) goes to an untranslated application address. During translation, the translator replaces each CTI with an exit stub that "re-enters" the translator for a new address, called a *trampoline*.

To safely re-enter the translator, the application's context must be saved in order to free registers for use by the translator, i.e., a *context switch* is done. The translator checks if there is translated code in the fragment cache (F$) for the requested application address. If so, the application context is restored and the corresponding translated code is executed. Otherwise, the dynamic translator builds a new set of translated instructions, called a *fragment*, and saves the fragment in the F$. When translation stops, the application context is restored and the new fragment is executed.

To avoid unnecessary context switches, *fragment linking* overwrites trampolines with a direct CTI to their target fragment [4]. This is not possible for indirect CTIs since their targets may change during execution. Instead, the *Indirect Branch Translation Cache (IBTC)* is used [10].

2.2 Tuning DBT for the ARM

To get low overhead from DBT, the translator must take into account instruction set nuances. For example, the cost of saving the `eflags` register on the Intel IA-32 architecture leads to certain implementation choices [10, 19]. The microarchitecture and caches are also important. For example, the way branches are handled by the AMD Opteron 244 leads to different DBT implementation choices than the Intel Pentium IV Xeon [10]. The ARM is like other processors in this regard. Indeed, the ARM's instruction set and exposed PC lead to complications and interactions with the caches and TLBs.

Parameter	Value	Interaction
1. IBTC size	Typical range 512 to 32K entries	Larger IBTC: better hit rate, increased D$ and D-TLB pressure
2. Reprobing	Disabled or enabled	Enabled: Better IBTC utilization, introduces search overhead
3. Reprobe depth	Typical range 1 to 3	Larger depth: Better IBTC utilization, increases overhead on a miss
4. Lookup placement	Inline or out-of-line	Inline: Lower instruction count, increases I$ pressure
5. Table placement	Separate location or next to lookup	Separate: Higher instruction cost
6. Fast checked returns	Enabled or disabled	Enabled: Introduces address check overhead, may avoid IBTC lookups

Figure 3. Configuration parameters for writes to the exposed PC

2.2.1 Reads of the Exposed PC

To handle reads of the exposed PC, we use emulation: the original PC of the untranslated instruction is made available to the translated instruction. Figure 2 shows how to handle reads of `pc`[2].

The first case shows an `add` instruction that has `pc` and a constant as source operands. During translation, the translator folds the computation: it has the PC of the instruction at translation time and the constant (from the instruction). It emits the folded value into a scratch slot (labelled `Value`). The `add` instruction is changed to a load of the folded value into register `r0`.

The second case is a subtract with two register operands, one of which is `pc`. In this situation, the computation cannot be folded because `r1`'s value is unknown until run-time. Instead, the original value of `pc` is emitted into a scratch slot (labelled `Value`) and loaded into the destination register before the subtract.

The final case shows a situation where the destination register is also a source register, so it cannot be used to hold the original value of `pc`. Instead, another register (`r1` in the example) is spilled to hold the value for the computation, and reloaded at the end of the emulation code.

PC-relative loads and stores with immediate offsets are handled specially. Data values (e.g., large constants) must be placed relatively close to the instructions that reference them, due to the ARM's maximum 4KB immediate offset in load and store instructions. A translated instruction, having been brought into the F$, is often too far away from the original referenced data to access it directly. The DBT can instead generate an "address pointer" for the PC-relative load (or store). The pointer is the address of the original referenced data in the application code and is placed near the translated instruction (i.e., within 4KB). When executing, the pointer is first loaded into a register and then dereferenced by another load (or store) to access the original data. This approach emulates what the application code would have done.

2.2.2 Writes to the Exposed PC

A write to the exposed PC is essentially an indirect CTI. Because the target application address of an indirect CTI is unknown until the control transfer executes, the dynamic translator has to map the application address to a F$ address (translated code) every time the indirect CTI executes. Indirect CTIs can be handled by returning to the translator to map their target address to a F$ address. However, this solution is too expensive due to the run-time overhead of context switches. Instead, the IBTC can be used to reduce the cost.

The IBTC is a hash table that holds mappings from indirect CTI target application addresses to F$ addresses. The translated code for the application address is located at the target F$ address. When an indirect CTI is translated, code is emitted to access the IBTC. If a target address is not in the IBTC (i.e., a miss), the translator is entered to map the address. If a fragment does not exist for the application address, the instructions at the address are translated. A single IBTC is typically shared by multiple indirect CTIs [10]. The IBTC has parameters that influence its effectiveness.

Figure 3 summarizes the possible IBTC configuration choices. Each choice has a trade-off between instruction count (IC) and interaction with the caches and TLBs. The first to fourth choices are standard ones from Hiser et al. [10]. The fifth and sixth parameters are specific to our ARM implementation.

The *table placement* parameter, the fifth in Figure 3, determines where to put the IBTC table. The table can be in data memory, and therefore accessing the table requires loading its address from memory. This approach can lead to increased memory contention. Alternatively, the table can be adjacent to the IBTC lookup code, which allows the use of PC-relative accesses to the table. For a shared IBTC, this option should be enabled only with an out-of-line lookup (OLL).

While the IBTC is effective at handling general indirect CTIs, return instructions have a property that can be used to reduce their overhead. A return transfers control to an application address that can be mapped *prior* to making a call. We developed a technique for the ARM to exploit this property, called "fast checked returns" (FCR). This parameter is the final one in Figure 3. FCR differs from previous fast return implementations [15] in that it checks that return addresses are within the fragment cache.

Unlike a normal return in the application code, a translated return does not go to the instruction after the translated call. Because a new fragment is formed for an address that is the target of a return, the translated return address can be anywhere in the F$. FCR requires coordination between calls and returns to pass the correct translated return address. A call is translated in two steps. First, the target of the call is inlined at the call site. Second, the side effects of the call are emulated. One side effect sets the ARM's link register (the return address register). FCR sets the link register to the translated return address in the emulation of the call. If the code at the application return address is not translated, it is pre-emptively translated to obtain the needed F$ return address.

There is a complication to this approach: The ARM lacks explicit call and return instructions. The dynamic translator must speculate which indirect CTIs are calls/returns. Calls are relatively easy to predict: they are typically done with a "jump and link" instruction. Returns are more difficult to predict. A return can, in fact, be any instruction that sets the exposed PC. Furthermore, the link register can be manipulated between a call and return.

Due to these problems, an instruction that is predicted as a return can not simply jump to the value in the link register. If the prediction is correct and the link register was not manipulated between the call and return, then control will transfer to the correct translated address. If the prediction is wrong, a call was miscategorized, or the link register was changed, then control can be "lost" by the dynamic translator. To deal with this problem, FCR uses an "address check" prior to executing an indirect CTI. When the translator recognizes a call (jump and link), it sets the link register to the translated return address. On an indirect CTI (which may be a return), a fast check is done to determine whether the destination ad-

[2] The `pc` register always contains the address of the current instruction + 8.

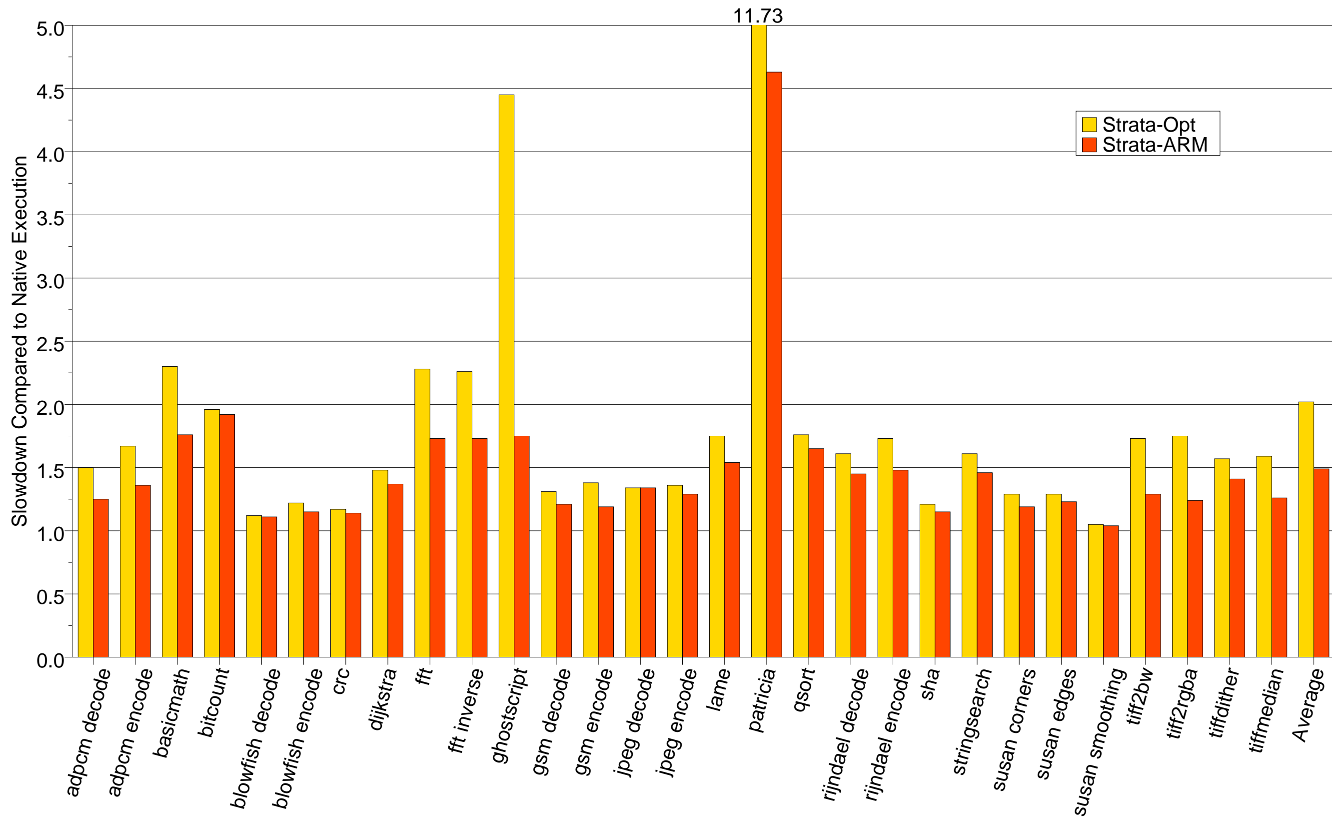


Figure 4. Program slowdown with Strata relative to native execution

dress is in the F$ (whose bounds are known) or the application text. If the value is a F$ address, then the indirect CTI is a return whose corresponding call instruction was detected. In this case, control can transfer to the address without an IBTC lookup. If the value is an application address, then the indirect CTI may not be a return or perhaps, the translator miscategorized a call. The IBTC is used to map the address in this situation.

Simple manipulations of the return address register (e.g., in library code) can be easily tracked and FCR disabled when they are discovered. More complicated manipulations could thwart the FCR mechanism but is unlikely to happen in normal program code. The MiBench programs and associated library code do not have any such manipulations that we have encountered.

2.3 Experimental Results

To identify the best choices for the configuration parameters, we experimentally tried a large range of values. We compared the best settings to recommendations for a general-purpose system to understand how the configuration influences overhead.

2.3.1 Methodology

We retargeted the Strata DBT [16] to the ARM/Linux platform. Strata is a reconfigurable and retargetable infrastructure that has been used for many purposes in embedded systems, including code compression [17], code dissemination [22], code security [11] and software instruction caching [3].

Strata is configured to form dynamic basic blocks [18], which minimizes code growth (important for embedded systems) [2]. An effectively unbounded code cache is used. Experiments were run on a Gumstix Verdex XL6P embedded system, which has 128 MB memory and a 600MHz Intel/Marvell XScale PXA270 ARM processor with 32KB I$ and D$. We used Linux 2.6.21, and GCC 4.1.1 with "-O3".

We used the MiBench benchmarks for embedded systems [8]. *pgp.decode*, *pgp.encode*, *rsynth*, *sphinx*, and *typeset* did not run in our setup. The large data sets were used for most programs. However, the large data sets for *adpcm.decode*, *adpcm.encode*, *tiff2bw*, *tiff2rgba*, *tiffdither*, and *tiffmedian* did not fit in the device's memory. The small input data sets were used instead for these programs. Because many MiBench programs execute extremely quickly, we modified them to execute for at least one second (approximately) by changing them to call their main function multiple times. These programs are *adpcm.decode*, *adpcm.encode*, *blowfish.decode*, *blowfish.encode*, *crc*, *dijkstra*, *gsm.encode*, *jpeg.decode*, *jpeg.encode, sha*, *stringsearch*, *susan.corners*, *susan.edges*, *susan.smoothing*, *tiff2bw* and *tiffdither*. This modification ensures that the impact of the IBTC and other control mechanisms can be observed for programs with fast execution times. Otherwise, program run-time is dominated by time in the *translator*, rather than time in the *translated code*. Finally, the programs and data sets were placed in a RAM disk and all programs were run with a light system load and the network driver disabled. This setup ensures accurate timing runs. Performance results are reported based on user time.

2.3.2 Initial Performance Overhead

To find the best configuration parameters, we did a series of experiments that tuned the translator by trying all combinations of parameters from Figure 3. The typical values listed in the figure were used for IBTC size and probe depth.

Program	Strata-ARM Overhead	IC Ratio	Instruction Distribution	Instr. Miss Distribution	Data Miss Distribution
adpcm.decode	**125.0**	**104.0**	**49.3**	**25.8**	**25.0**
adpcm.encode	**136.0**	**102.4**	**46.0**	**39.3**	**14.7**
basicmath	**176.0**	**137.4**	**85.0**	**11.5**	**3.5**
bitcount	**192.0**	**141.2**	**89.8**	**8.1**	**2.1**
blowfish.decode	111.0	112.1	83.2	5.4	11.4
blowfish.encode	115.0	105.8	84.8	5.3	10.0
crc	114.0	115.3	89.2	6.1	4.7
dijkstra	137.0	130.5	87.9	7.2	4.8
fft	**173.0**	**143.3**	**87.3**	**10.0**	**2.7**
fft.inverse	**173.0**	**143.4**	**87.1**	**10.2**	**2.7**
ghostscript	**175.0**	**135.8**	**78.6**	**15.5**	**5.9**
gsm.decode	121.0	113.8	86.7	9.3	4.0
gsm.encode	119.0	104.4	82.5	14.8	2.7
jpeg.decode	**134.0**	**107.2**	**65.2**	**12.8**	**22.0**
jpeg.encode	129.0	118.6	76.0	11.6	12.5
lame	**154.0**	**129.9**	**85.4**	**10.6**	**3.9**
patricia	**463.0**	**156.5**	**45.1**	**51.2**	**3.7**
qsort	**165.0**	**140.4**	**86.6**	**8.7**	**4.7**
rijndael.decode	**145.0**	**108.7**	**82.8**	**10.1**	**7.1**
rijndael.encode	**148.0**	**112.2**	**82.2**	**10.9**	**6.9**
sha	115.0	124.1	88.2	6.9	4.9
stringsearch	146.0	127.1	65.2	9.0	25.7
susan.corners	119.0	113.3	72.6	10.3	17.1
susan.edges	123.0	118.4	83.9	7.7	8.4
susan.smoothing	104.0	111.6	89.3	5.6	5.0
tiff2bw	129.0	121.6	68.6	8.4	23.0
tiff2rgba	124.0	140.2	54.3	9.6	36.1
tiffdither	141.0	127.1	78.5	14.0	7.5
tiffmedian	126.0	142.6	70.6	12.3	17.1
Average	149.0	123.8	N/A	N/A	N/A

Figure 5. Profile of relative distribution of total execution time

From this large set of experiments, we found the *best configuration* that worked well over most benchmarks. We call this configuration "Strata-ARM." We do not report full results from the experiment for brevity. We compared Strata-ARM to the recommended configuration from Hiser et al. [10] (called "Strata-Opt"). This configuration has been shown to have very low run-time overhead on Intel IA-32 desktop systems. On the SPEC2000 benchmarks, Strata-Opt has a 4.5% overhead for the Pentium IV Xeon [10]. Strata-Opt has a large 32K entry IBTC with inline lookup placement. Strata-ARM has a 8K IBTC, with reprobing (depth 2), out-of-line lookup, PC-relative table, and fast checked returns.

Figure 4 shows slowdown for Strata-Opt and Strata-ARM. The slowdown is relative to "native execution" (without Strata). The slowdown for Strata-Opt varies from 1.05 (*susan.smoothing* to 11.73 (*patricia*) with an average of 2.02. This run-time overhead shows the impact of the ARM instruction set and certain microarchitecture features in comparison to Strata-Opt for the Intel Pentium IV Xeon [10].

By tuning Strata's configuration, the slowdown is much improved. Strata-ARM has a slowdown of 1.04 (*susan.smoothing*) to 4.63 (*patricia*), with an average of 1.49. Several benchmarks have good improvements with Strata-ARM (e.g., *dijkstra*, *fft* and *ghostscript*). *patricia* does the best: Its slowdown is reduced from 11.73 to 4.63. The improved slowdowns are due to FCR (lower IC) and a small IBTC with OLL (less I$ pressure).

2.3.3 Hardware Performance Monitoring

Although Strata-ARM demonstrates that tuning to the ARM is beneficial, the results show that the overhead should be improved further. To find the source of the overhead, we profiled the applications using hardware performance counters. The PXA270 has counters for cache misses, TLB misses, instruction fetches, and other events.

Figure 5 summarizes the collected profile information. We used the PXA270 hardware counters to collect: total cycles, instruction count, number of data fill buffer stalls, number of cycles spent on data fill buffer stalls, and number of instruction fetch stalls. The second and third table columns compare program slowdown (expressed as percentage from Figure 4) and instruction count ratio. IC_{ratio} is $(IC_{strata}/IC_{native} * 100)$. IC_{strata} is a program's IC when run with Strata-ARM and IC_{native} is the IC when executed natively. The remaining columns show the distribution of execution time for a program among instruction execution, instruction fetch stalls and data access stalls. The values are percentages of the total cycle count (e.g., *adpcm.decode* spends 49.3% of its total cycles on instruction execution).

A comparison between slowdown and IC ratio hints why performance is harmed by DBT. Of course, such a comparison is only approximate because some instructions take many cycles (e.g., load-multiple) and performance counters can suffer from measurement error. Nevertheless, it gives insight into overhead. For example, *adpcm.decode* has a 125% overhead and 104% IC ratio. The difference between these values suggests that *adpcm.decode*'s overhead *does not* arise from executing more instructions with DBT. Instead, the overhead is likely due to adverse interactions with the caches. Another example is *crc*, in which the overhead and IC ratio are similar. This program's overhead is due to code expansion from DBT. A few programs, such as *sha*, have better overhead than IC ratio.

This difference is likely due to out-of-order execution among instructions in the PXA270's separate integer and memory pipelines.

On average, IC ratio is 123.8%, which indicates that the code expansion from Strata-ARM is small. However, the average overhead is 149%, which suggests that slowdown can be lowered, if cache and TLB interactions can be mitigated. Techniques that mitigate the interactions will be most effective for programs with a moderate to large difference between IC ratio and slowdown. In Figure 5 the programs in bold have an IC ratio and slowdown that differ by at least 20%. For brevity, we focus on these programs. Final results are reported for all benchmarks.

From the last three table columns the source of interactions can be identified. For example, *ghostscript* spends 78.6% of total cycles on instruction execution, 15.5% on instruction fetch stalls, and 5.9% on data access stalls. The results reveal why *patricia*'s slowdown is poor: It spends 51.2% on instruction fetch stalls and 3.7% on data access stalls. The percentage of cycles on instruction fetch stalls is very high in this program in comparison to the others. *adpcm.decode* and *adpcm.encode* are also interesting programs because a large part of their total cycles is spent on data access stalls. These programs apparently achieve an inherent benefit of DBT: Translated code is arranged in the F$ based on execution paths, which improves I$ locality. In turn, relatively more time is spent on other events (i.e., data access stalls). Comparing IC ratio and slowdown for *adpcm.encode* supports this observation: It has a 102% IC ratio and 136% overhead.

As these results show, both instruction fetch stalls and data access stalls are important sources of overhead arising from DBT. In some programs, data access stalls account for more overhead than instruction fetch stalls (e.g., *jpeg.decode*). In others, instruction fetch stalls dominate (e.g., *patricia*). Stalls are a consequence of increased pressure on the caches and TLBs (i.e., the cache and TLB miss rates increase).

3. Data Value Handling

The most likely source of the increase in cache and TLB pressure is the way that PC-relative loads and stores in the application code, as well as ones dynamically generated, are handled by DBT. PC-relative addressing is used for access to data values and constants. Such operations are used for 1) native application constants (generated by the compiler); 2) constants in translated code (generated by the translator); 3) "spill" locations for emulation (see Figure 2); and, 4) data structures used to invoke the translator. We classify these PC-relative operations into two categories: data values used in the translated code and data values needed to invoke the translator. We refer to both data values and constants as simply "values."

3.1 Improving Data Behavior in the Translated Code

Strata-ARM leaves an application's "native constants" in their original locations and uses indirection to reach them, with an address pointer. We call this approach "address emulation." Address emulation needs two memory accesses for a native constant targetting two separate virtual memory pages. This increases IC and leads to cache/TLB pressure.

The handling of a native constant can be improved by copying it to a location next to the translated code that references it. The offset in a PC-relative load is modified, when translated, to reference the new location. To ensure that there are no writes to an apparently constant value that is moved, the original memory page can be set read-only. A write to the original location will cause a trap. If a trap occurs, the translated code can be flushed and regenerated with address pointers to leave constants in place[3]. Similarly, spill locations can also be emitted near the translated code that references them.

[3] This does not happen in MiBench.

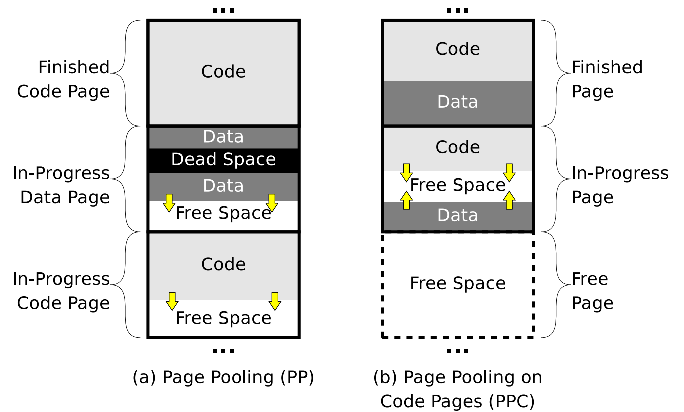


Figure 6. Page pooling and page pooling on code pages

To avoid executing values as instructions (intermixed with code), a branch must be inserted to jump around them. This branch adds to IC and may also contribute to I$ pressure.

Alternatively, the data values and constants needed by a fragment can be "pooled." We call this technique "fragment pooling" (FP). Grouping values creates better locality, which leads to better cache and TLB behavior.

To pool values, during fragment formation values are placed in a queue as they are encountered or needed for emulation. When fragment formation is finished, the values are processed. The value pool is emitted after the last instruction in the fragment (i.e., after a fragment's trampolines) and offsets in PC-relative accesses to pooled values are fixed. Because the pool is after the trampolines, a branch around it is unnecessary. FP splits large fragments (more than 1024 instructions) to ensure that the pool can be reached by the PC-relative loads and stores in the fragment.

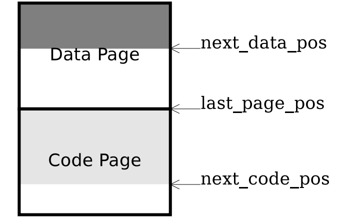
Although FP reduces memory pressure for native constants (versus address emulation), it can still lead to unnecessary cache and TLB pressure. Value pools at the end of each fragment effectively create "holes" between two fragments' code, which can harm instruction locality when control flows from one fragment to the next (adjacent) one. Similarly, the code between value pools harms data locality. It is also likely that both I-TLB and D-TLB entries will be allocated for the same virtual memory page, which increases overall TLB pressure. Depending on F$ layout, translated code execution may span many memory pages. In such situations, the increased TLB pressure from each fragment can lead to overall decreases in performance due to TLB misses.

A better way to arrange the pool is to group values *across fragments* in a contiguous memory region, which can potentially improve cache locality and TLB utilization. We developed two techniques to pool data in larger granularities than a fragment: "page pooling" (PP) and "page pooling on code pages" (PPC). Because the ARM has only a $\pm$4KB offset range, the values must be within this range. PP and PPC group data on 4KB virtual memory pages and use PC-relative accesses to pooled values.

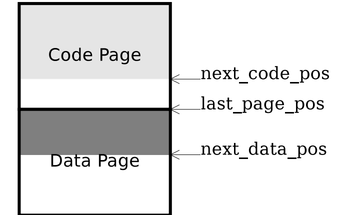
Figure 6 shows how PP and PPC differ. PP puts instructions and data on separate code and data pages, as shown in Figure 6(a). The two page types are interleaved. A value needed by an instruction is placed on an adjacent data page (before or after the code page), depending on the offset range (i.e., which page is reachable from the instruction). PP has the advantage that the TLB is better utilized as only one TLB entry (I-TLB *or* D-TLB) is needed to map a page. However, PP causes "dead space" on a data page because not all locations are likely to be needed or can be reached by instructions.

```
1   OnStartup()
2       next_data_pos←start of fragment cache
3       next_code_pos←next_data_pos+PAGE_SIZE
4       last_page_pos←next_code_pos
5   AdjustPointers()
6       if Abs(next_code_pos−next_data_pos)>MAX_OFFSET
7           if next_code_pos>next_data_pos
8               next_data_pos←next_code_pos−MAX_OFFSET
9               if next_data_pos ≥ last_page_pos
10                  next_data_pos←last_page_pos+PAGE_SIZE
11                  last_page_pos←next_data_pos
12          else
13              while next_code_pos−next_data_pos>MAX_OFFSET
14                  Emit(next_code_pos,NOP)
15                  next_code_pos←next_code_pos+4
16  AfterTranslateInstruction(fragment,tramp_space)
17      if (next_code_pos<next_data_pos) and (last_page_pos−next_code_pos≤tramp_space)
18          next_PC←EndEarly(fragment)
19          EmitTrampolines(fragment)
20          next_code_pos←last_page_pos+PAGE_SIZE
21          last_page_pos←next_code_pos
22          Enqueue(next_PC)
```

(a) Algorithm



(b) Data before code



(c) Code before data

Figure 7. Page pooling (PP) algorithm

Figure 6(b) shows that PPC puts instructions and data on the *same* page, but pools constants and data values in a contiguous memory region. Code starts at the beginning of the page and grows from a low to high address. Data starts at the end of the page and grows from a high to low address. The figure shows an "in-progress page" where code and data have partially filled a page. When there is no room for code or data on a page, the construction of a fragment is stopped. It resumes on the next available page. Because a PC-relative instruction and its associated value are on the same 4KB page, the value is always reachable. PPC does not have unused memory locations as a result. The lack of "dead space" improves footprint and possibly D$ and TLB behavior. However, the approach suffers from the same issue as FP, with separate I-TLB and D-TLB entries that map the same pages.

With both page pooling techniques, the end result is that groups of fragments that are temporally related have their constants spatially grouped. That is, when fragments are translated, they are placed one after another one in successive order (along the initial execution path). Thus, a group of fragments that are translated along a path will appear next to one another in the F$. Indeed, the data values associated with the fragments will have a similar layout.

Figure 7(a) shows the algorithm for PP. When the translator starts-up, the algorithm sets start pages for data and code (lines 1-4). During translation of an application instruction, it may be necessary to emit a data value. When the distance between the data and the instruction that references it cannot be encoded in the offset field of the instruction, the pointers to the next code and data slots must be adjusted (lines 5-15). There are two possibilities. If the code uses data in the previous page, as shown in Figure 7(b), the pointer to the next data slot must be increased to a reachable position, which leaves some dead space in the data page (lines 7-8). If that position exceeds the data page limit, a new data page must be allocated after the code page (lines 9-11), shown in Figure 7(c). If the code cannot reach the data in a following page, the adjustment is done by increasing the pointer to the next instruction slot (lines 13-15). When spilling a register, the translator ensures that the two instructions which save and restore the value use the same data slot.

After translating an application instruction, some clean up is needed. The translator checks whether the code is about to overflow into an occupied data page (line 17). In this case, the current fragment is finished early (lines 18-19). The current fragment (prematurely ended) will be linked to its continuation fragment (requested on line 22). Because a code page holds 1,024 ARM instructions, it is uncommon for fragments to end early.

PPC is similar to PP, but it must handle data values on the same page as the fragment. Therefore, no dead space in code or data is created. However, it must always check for a future page overflow after the completion of a fragment (PP does this check only when the data page is after the code page).

3.1.1 Experimental Results

We implemented FP, PP, and PPC in Strata-ARM to investigate their effect on overhead. Figure 8 shows slowdown relative to native execution for the programs in bold from Figure 5.

Several benchmarks have much improvement versus Strata-ARM. For example, *rijndael.encode* had a 1.48 slowdown with Strata-ARM. With the pooling techniques, its slowdown is improved to 1.38 with FP (bar labeled "Strata-ARM+FP"), 1.3 with PP (bar labeled "Strata-ARM+PP") and 1.2 with PPC (bar labeled "Strata-ARM+PPC"). *adpcm.decode*, *adpcm.encode*, *ghostscript*, *jpeg.decode*, *patricia*, and *rijndael.decode* also had noticeable improvement. The other programs had a negligible (at most, a few percent) or no performance improvement with the pooling techniques.

patricia had the largest gain: its 4.63 slowdown with Strata-ARM was improved to 2.43 with Strata-ARM+PPC. *adpcm.encode* is particularly interesting. Pooling constants and data values improved the TLB and cache locality enough that the program achieved a speedup. It had a 1.36 slowdown with Strata-ARM, which was improved to 1.04 with FP, 0.96 with PP (speedup) and 0.88 with PPC (speedup).

Overall, PPC gives the best improvement when page pooling is effective. Although this approach mixes code and data on a page, avoiding dead space (versus PP) provides enough locality benefit to make this technique the best choice.

3.2 Improving Data Behavior for the Translator

When the dynamic translator is re-entered from the translated code, information has to be passed to it. The information is a F$ address (`from-fragment`) and a requested application address (`to-address`). On the ARM, this information must be loaded (as

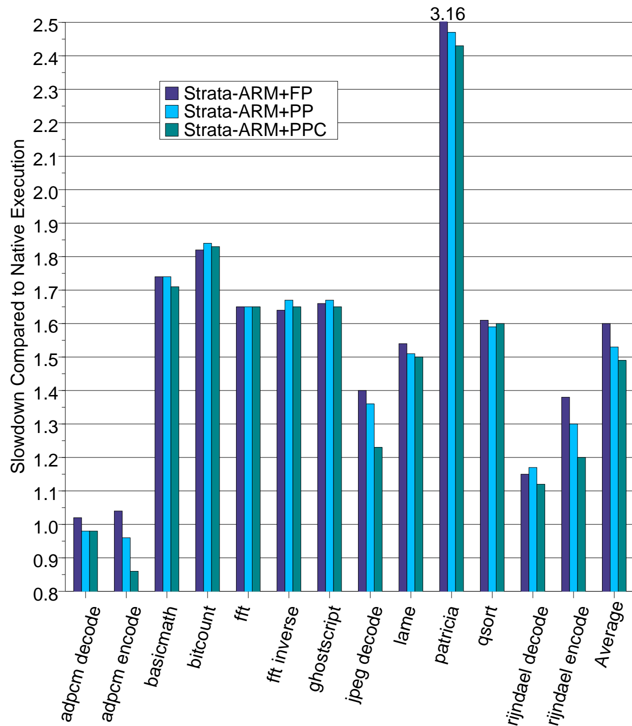


Figure 8. Slowdown with pooling (FP, PP, PPC)

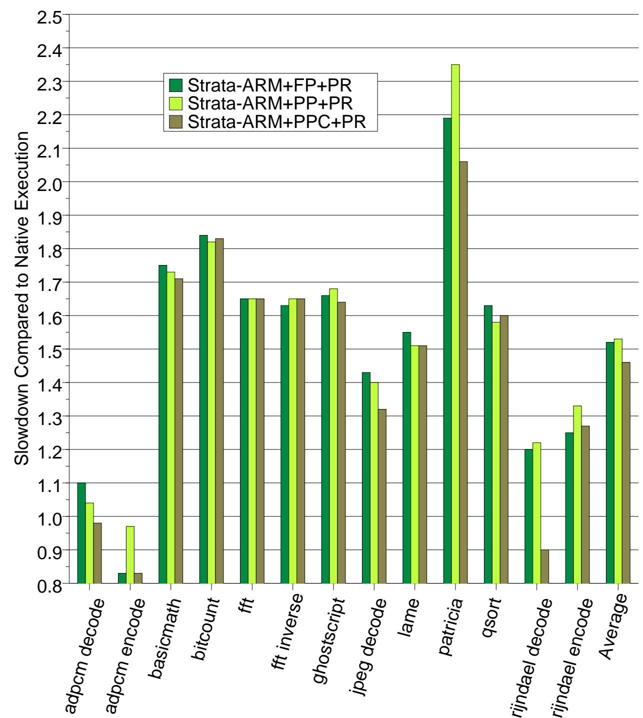


Figure 9. Slowdown with paged re-entry

large constants) and put in argument registers for the translator. Arguments passed to the translator are set in the fragment trampoline.

The `from-fragment` and `to-address` values are stored in the trampoline and loaded with PC-relative operations. With this approach, constants needed to enter the translator are spread throughout the F$. Furthermore, the trampoline is relatively large and once linked, will leave dead, unreachable instructions in the F$.

The locality of the re-entry process can be improved by grouping together arguments needed by fragments to re-enter the translator. The improvement in locality may be particularly important during program phase changes, when the translator is re-entered rapidly in succession to translate new code. However, even if the `from-fragment` and `to-address` values are pooled (e.g., with FP, PP, or PPC), the trampoline is still large due to setup of the argument registers.

A better approach, which we call "paged re-entry" (PR), both pools arguments and reduces trampoline size. PR groups `from-fragment` and `to-address` arguments in a pool, called the "argument table." This table is easily accessed by the translator and is kept separate from data values and translated code. Upon re-entering the translator, an index to an entry in the argument table is passed rather than the arguments themselves. On the ARM, an index small enough to fit in the 8-bit immediate field can be used. This simple table structure helps data locality.

To keep the argument table's index small enough to fit in an 8-bit immediate (avoiding load instructions), only 256 trampolines can be "live" at any one time. In many programs, 256 trampolines are sufficient. However, some programs have more than 256 live trampolines and therefore need a larger index. To handle this situation, PR tracks the number of live trampolines and increases the index as necessary in quantities of 256. Rather than encoding large indices as PC-relative constants (or small immediates with shifts), PR uses multiple re-entry points (i.e., a jump table) into the translator. Each re-entry point selects a different "page" of 256 entries in the argument table.

Indices are initially assigned in increasing order. As fragments are linked, trampolines are patched and their corresponding entries are added to a LIFO free list. The free list is built into the argument table to save space. A reclaimed entry is reused when a new trampoline is created. This ensures good locality and minimizes the size of the argument table.

A single TLB entry can map many argument entries. During rapid successive invocations of the translator (i.e., during a program phase change), a single TLB entry is likely to live across context switches (i.e., between execution of the translated code and the translator), reducing TLB misses. A single TLB entry maps 512 argument table entries (each argument table entry is 8 bytes), which is sufficient for most programs. Similarly, a D$ entry may live across multiple invocations.

Paged re-entry inherently reduces trampoline size because it passes only one argument to the translator (the table index). The index, being encoded as an immediate, avoids a load of a large constant. As a result, a trampoline built with PR is short (three instructions). After fragment linking, there is less dead space between fragments, improving I$ locality. However, IC is increased slightly due to the multiple re-entry points and argument table lookup.

3.2.1 Experimental Results

We implemented paged re-entry in Strata-ARM with FP, PP and PPC. Figure 9 shows the slowdowns of the benchmarks with PR. The figure shows that PR can lead to a reduction in slowdown.

Typically, the best results are achieved when PR is used with PPC. For example, *adpcm.encode*'s slowdown with Strata-ARM+PPC is reduced by a small amount with Strata-ARM+PPC+PR. *patricia* is a case where there is a large improvement. Strata-ARM+PPC had

a slowdown of 2.43, which is lowered to 2.06 with paged re-entry. *rijndael.decode* also has a significant reduction with PR and PPC.

Many programs do not have an improvement with PR. The lack of an improvement is due to a relatively small number of translator invocations. PR lowers overhead only when the re-entry process originally caused noticeable overhead in a program's execution. In some cases, PR can actually cause slowdown (e.g., *jpeg.decode* and *rijndael.encode*). The degradation is due to a larger instruction count with paged re-entry. Nevertheless, when PR is beneficial, it can lead to a large gain and we conclude that it should be enabled (i.e., on average, the slowdown is reduced with PR).

4. Overall Results

Figure 10 gives a comparison of slowdown for all programs with Strata-Opt, Strata-ARM, and Strata-ARM+PPC+PR. The figure shows that carefully selecting configuration parameters is important to reduce slowdown. For example, Strata-Opt has a 2.28 slowdown on *fft*, which is decreased to 1.73 by Strata-ARM. *ghostscript* is another program that has much improvement (Strata-Opt's slowdown of 4.45 is improved to 1.75).

Strata-ARM+PPC+PR achieves a large reduction in slowdown versus Strata-ARM for several programs (e.g., *adpcm.encode*, *patricia*, *rijndael.decode*, and *rijndael.encode*). These programs are the ones that profiling indicated had the most pressure on the caches and TLBs. Other programs have a smaller but noticeable improvement. For example, *bitcount*'s slowdown is reduced from 1.92 to 1.83, *dijkstra*'s slowdown is reduced from 1.37 to 1.23, and *tiffmedian*'s slowdown is reduced from 1.26 to 1.15. Strata-ARM+PPC+PR has a different code layout than Strata-ARM, which may slightly hurt performance (e.g., *susan.corners*, *susan.edges*, *tiff2bw*, *tiff2rgba*).

From these results, we conclude that page pooling on code pages and paged re-entry provide significant benefit when dynamic translation puts much pressure on the caches and TLBs due to constants and data values. Although our results are specific to a particular ARM microarchitecture, we expect similar trends for other implementations. The absolute gains may differ somewhat based on cache and TLB capabilities. Our techniques further enable the use of DBT in ARM processors to provide many desirable capabilities, such as secure code execution and resource management.

5. Related Work

DBT is important because it can accomplish tasks that are not otherwise easily achieved. For this reason, DBT has attracted attention for embedded systems [2, 3, 5, 6, 9, 17, 22]. Several compelling uses of DBT in embedded systems have been explored. Shogan and Childers [17] describe a software-based code compression/decompression system built with dynamic binary translation. Zhou et al. [22] present an effective approach to disseminate code to wireless-enabled memory-constrained embedded devices (e.g., smartcards) with dynamic translation. Other uses include power management [21], program instrumentation [9], code security [12], and incremental loading [3]. Our techniques can help reduce overhead in any of these DBT uses. They make the uses more practical in an embedded ARM-based device.

Other research addressed the management of the code cache (i.e., the F$) for DBT in embedded systems. Guha et al. [6] developed techniques to reduce the size of trampolines in the translated code. Baiocchi et al. [2] examined the sources of code expansion in a dynamic translator for embedded systems with scratchpad memory and describe approaches to minimize the expansion. They also developed methods to avoid expensive code retranslation by compressing translated code [3]. This past work focused on how to effectively utilize code cache space and avoid eviction of important code. Our value pooling techniques are a form of code cache management: The techniques arrange data and code to improve locality and reduce TLB pressure, rather than reduce eviction and/or retranslation cost. Therefore, our techniques provide performance benefits whether or not an unbounded code cache is being used.

Ruiz-Alvarez and Hazelwood [14] comprehensively used performance monitors and simulation to evaluate the effects of two dynamic binary instrumenters on the caches and TLBs of x86 systems. Their findings agree with ours: DBT significantly and negatively affects both caches and TLBs. However, techniques to reduce these effects are not given. We provide novel approaches that mitigate the performance impact of DBT on caches and TLBs. Maebe et al. [13] built a dynamic binary translator for x86 that handles the mixing of code and data, similar to what occurs in ARM binaries. Hazelwood and Klauser [9] built a dynamic binary instrumenter for ARM and evaluated its performance using SPEC2000 benchmarks and an earlier ARM implementation (StrongARM-1110).

6. Conclusion

This paper described how to handle the ARM's instruction set and microarchitecture challenges for dynamic binary translation. It described methods both to address the ARM's exposed PC and to carefully tune a dynamic binary translator to instruction set and cache/TLB subtleties. From hardware profiling, the sources of DBT overhead were identified. Based on this study, novel techniques were developed to improve cache and TLB locality, including fragment pooling, page pooling, page pooling on code pages, and paged translator re-entry. These techniques were implemented in the Strata DBT system. With a configuration that has been shown to be good for general-purpose systems, Strata had an average 2.02 slowdown on the MiBench benchmark suite. The same translator, carefully tuned for ARM and with the overhead reduction techniques enabled, had a 1.31 average slowdown (a 54% improvement over the general-purpose configuration).

Acknowledgments

Supported in part by NSF grants CCF-0811352, CCF-0811295, CNS-0720483, CCF-0702236, and CNS-0551492.

References

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006.

[2] J. Baiocchi, B. Childers, J. Davidson, and J. Hiser. Reducing pressure in bounded DBT code caches. In *Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, 2008.

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

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Int'l Conf. on Programming language design and implementation*, 2000.

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

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

[7] S. Gupta, P. Pratap, H. Saran, and S. Arun-Kumar. Dynamic code instrumentation to detect and recover from return address

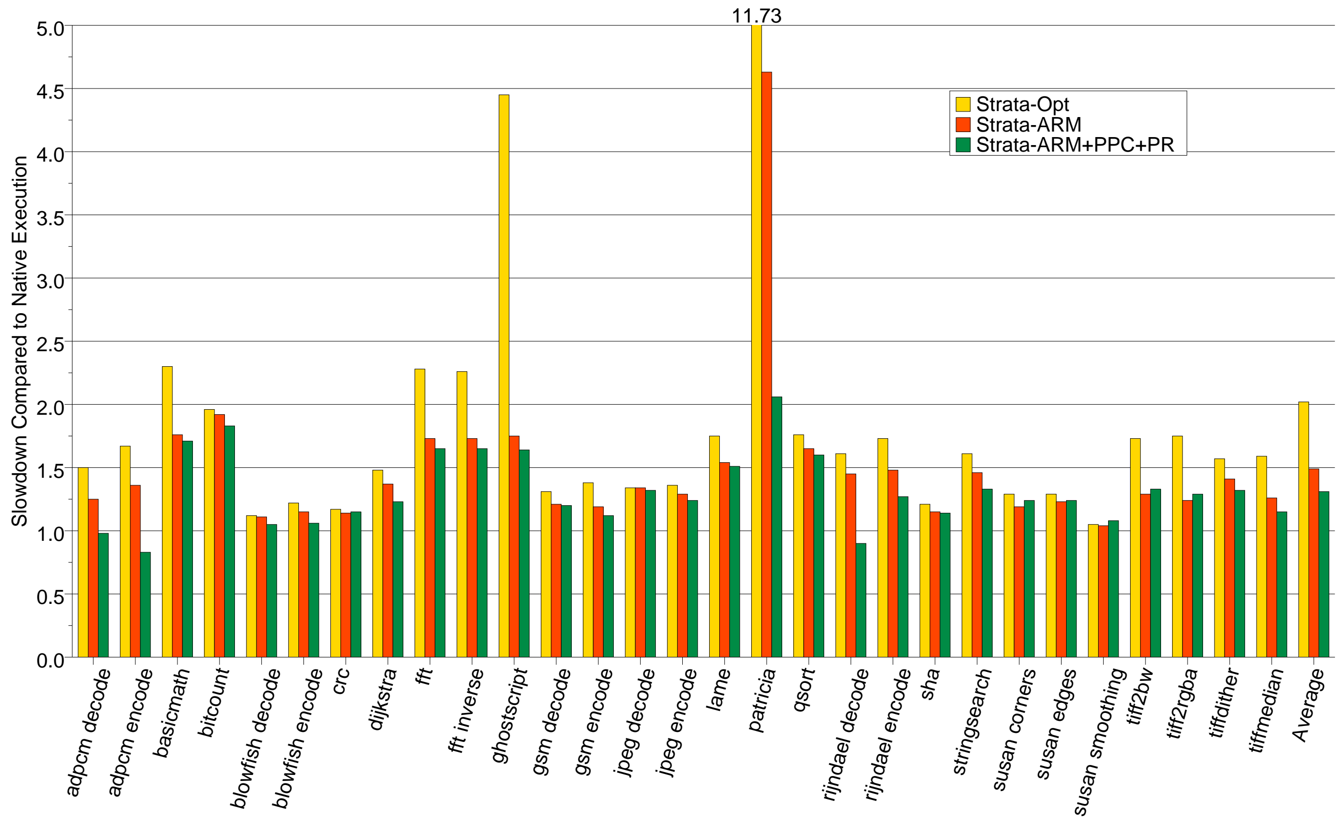


Figure 10. Slowdowns of Strata-Opt, Strata-ARM, and Strata-ARM+PPC+PR Compared to Native Execution

corruption. In *Intl'l Workshop on Dynamic Systems Analysis*, 2006.

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

[9] K. Hazelwood and A. Klauser. A dynamic binary instrumentation engine for the ARM architecture. In *Int'l. Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, 2006.

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

[11] W. Hu, J. Hiser, D. Williams, A. Filipi, J. Davidson, D. Evans, J. Knight, A. Nguyen-Tuong, and J. Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Int'l. Conf. on Virtual Execution Environments*, 2006.

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

[13] J. Maebe, M. Ronsse, and K. De Bosschere. Diota: Dynamic instrumentation, optimization and transformation of applications. In *Compendium of Workshops and Tutorials Held in conjunction with PACT'02*, 2002.

[14] A. Ruiz-Alvarez and K. Hazelwood. Evaluating the impact of dynamic binary translation systems on hardware cache performance. In *IEEE Int'l. Symp. on Workload Characterization*, 2008.

[15] K. Scott, N. Kumar, B. Childers, J. Davidson, and M. L. Soffa. Overhead reduction techniques for software dynamic translation. In *Int'l. Parallel and Distributed Processing Symp.*, 2004.

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

[17] S. Shogan and B. Childers. Compact binaries with code compression in a software dynamic translator. In *Int'l Conf. on Design, Automation and Test in Europe*, 2004.

[18] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[19] S. Sridhar, J. Shapiro, E. Northup, and P. Bungale. HDTrans: an open source, low-level dynamic instrumentation system. In *Int'l Conf. on Virtual Execution Environments*, 2006.

[20] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks. Dynamic-compiler-driven control for microprocessor energy and performance. *IEEE Micro*, 26(1), 2006.

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

[22] S. Zhou, B. Childers, and M. L. Soffa. Planning for code buffer management in distributed virtual execution environments. In *Int'l Conf. on Virtual Execution Environments*, 2005.