A Dynamic Binary Instrumentation Engine for the ARM Architecture

Kim Hazelwood
University of Virginia

Artur Klauser
Intel Corporation

ABSTRACT

Dynamic binary instrumentation (DBI) is a powerful technique for analyzing the runtime behavior of software. While numerous DBI frameworks have been developed for general-purpose architectures, work on DBI frameworks for embedded architectures has been fairly limited. In this paper, we describe the design, implementation, and applications of the ARM version of Pin, a dynamic instrumentation system from Intel. In particular, we highlight the design decisions that are geared toward the space and processing limitations of embedded systems. Pin for ARM is publicly available and is shipped with dozens of sample plug-in instrumentation tools. It has been downloaded over 500 times since its release.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Code generation, Optimization, Run-time environments

General Terms

Languages, Management, Measurement, Performance

Keywords

Pin, Binary instrumentation, Dynamic translation, Embedded architectures

1. INTRODUCTION

Understanding the run-time behavior and bottlenecks of applications is extremely important to both software developers and end users. Software-based instrumentation tools facilitate this task by enabling transparent access to the processor and memory state after every executed application instruction. Instrumentation tool users can gather arbitrary statistics about the run-time actions of an executing application without disrupting the normal behavior of that application. The information gathered can then be used to locate inefficiencies or errors in the implementation.

More recently, researchers have taken advantage of the transparent nature of DBI frameworks to do more than just inspect program behavior. The same frameworks have been used to implement run-time optimizations, run-time enforcement of security policies, and run-time adaptation to environmental factors, such as power and temperature. Yet, almost all of this research has targeted the high-performance computing domain.

The embedded computing market is a great target for dynamic binary instrumentation. On these systems it is critical for performance bottlenecks, memory leaks, and other software inefficiencies to be located and resolved. Furthermore, considerations such as power and environmental adaptation are arguably more important than in other computing domains. Unfortunately, robust, general-purpose instrumentation tools aren't nearly as common in the embedded arena (compared to IA32, for example). The Pin dynamic instrumentation system fills this void, allowing novice users to easily, efficiently, and transparently instrument their embedded applications, without the need for application source code.

Pin currently works on four architectures (IA32, EM64T, IPF, and ARM) and four operating systems (Linux, Windows, FreeBSD, and MacOS), and in fact a large portion of the code base is platform independent. In this paper, we focus on the ARM[1] implementation running on Linux, and discuss its unique challenges and features. Indeed some features—such as limited memory—will test the limits of our trace selection and caching algorithms. Other features—such as explicit instruction cache flush requirements—actually simplify our support for self-modifying code.

Prior work has shown the performance overhead of the ARM implementation of Pin to be up to twice that of the IA32 implementation [23]. In this work, we document our experiences tuning the implementation to the point where it rivals the overhead for IA32.

The remainder of this paper will proceed as follows. In Section 2, we provide an overview of dynamic instrumentation and the Pin system in particular. We also highlight some of the features of the ARM architecture itself to provide a foundation for the rest of the paper. Section 3 delves into the implementation details of Pin for ARM, and discusses how we handle challenges such as indirect branches and self-modifying code. Next, in Section 4 we report the run-time performance of our implementation and Section 5 highlights the potential applications of our system. Finally, Section 6 presents related work, and Section 7 concludes.

2. BACKGROUND

Before delving into the implementation details of Pin for the ARM architecture, we first provide a high-level view of Pin and an overview of the ARM architecture.

CASES'06, October 23–25, 2006, Seoul, Korea.
Copyright 2006 ACM 1-59593-543-6/06/0010 ...$5.00.

[1] We will use the term ARM to refer to any processor that implements the ARM ISA, including StrongArm and XScale.

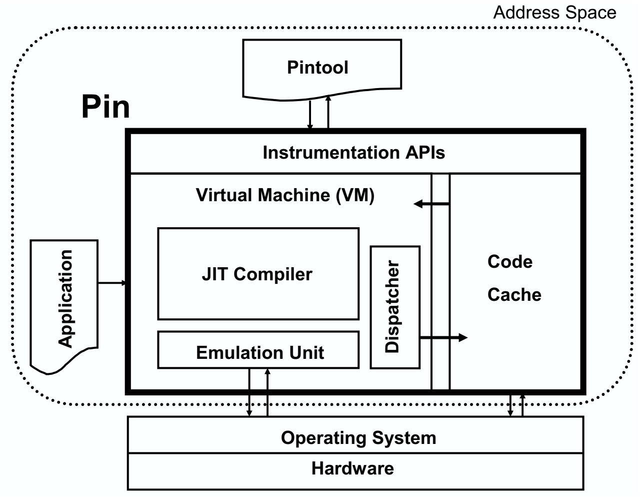


Figure 1: Software architecture of Pin.

2.1 Pin Overview

Pin [23] is a dynamic binary rewriting system developed at Intel Corporation. Pin was designed with instrumentation in mind. Hence, instrumenting a program is both easy and efficient. A user may write instrumentation tools using an API that is rich enough to allow many plug-ins to be source compatible for all the supported instruction sets. Pin allows a tool to insert function calls at any point in the program. It automatically saves and restores registers so the inserted call does not overwrite application registers.

Figure 1 illustrates Pin's software architecture. At the highest level, Pin consists of a virtual machine (VM), a code cache, and an instrumentation API invoked by Pintools. The VM consists of a just-in-time compiler (JIT), an emulator, and a dispatcher. After Pin gains control of the application, the VM's components work together to execute the application. The JIT compiles and instruments application code, which is then launched by the dispatcher. The compiled code is stored in the code cache. Entering/leaving the VM from/to the code cache involves saving and restoring the application register state. The emulator interprets instructions that cannot be executed directly, such as system calls that require special handling from the VM. Since Pin sits above the operating system, it can only capture user-level code.

Like many other binary rewriters, Pin uses a code cache to store previously instrumented copies of the application to amortize its overhead. Code traces–or more specifically, *superblocks*–are used as the basis for instrumentation and code caching in Pin. After generating a trace, Pin immediately places it in the code cache and updates the cache directory hash table. For every potential off-trace path, Pin generates an exit stub, which redirects control back to the VM and passes information about the next trace to execute. Over time, Pin will patch any branches targeting exit stubs directly to the target trace in the code cache, which greatly improves performance. A thorough description of the internal functionality of Pin is provided elsewhere by Luk et al. [23].

2.2 ARM Architecture Overview

ARM is an acronym for Advanced RISC Machines. Most implementations of the ARM architecture focus on providing a processor that meets the power and performance requirements of the embedded systems community. Unlike processors designed for high-performance computing, ARM-based processors often lack features such as a floating-point unit or a second- (or third-) level on-chip cache.

The ARM instruction-set architecture (ISA) is quite powerful. ARM is a RISC architecture with 32-bit fixed-width instructions. (In this paper we do not discuss Thumb, a 16-bit ISA extension to ARM.) The ISA provides such features as full generalized predication, the PC being a general register that can be read and/or written by any instruction, and single-instruction context save and restore support. For a dynamic binary rewriter, some of these features can be challenging to handle. In the following section we describe how Pin for ARM tackles some of these challenges.

3. IMPLEMENTATION DETAILS

Dynamic instrumentation systems perform various transformations to the original code sequences before inserting the transformed copy into the code cache. Some of these transformations are designed to ensure that the VM maintains control of execution at all times, and control never escapes back to the original, uninstrumented version of the executable. Other transformations ensure transparency to the executing application.

3.1 Application Transparency

Pin provides transparency to any application running under its control. All memory and register values, including the PC, will appear to the application as they would had the application been run directly on the hardware. Yet, in reality the code never executes at its original address but in the code cache instead. To achieve transparency, all original references to the PC are replaced by appropriate constants before insertion into the code cache. PC updates that can not be statically analyzed are converted into sequences that enter the VM at run time, which will inspect the new PC and transfer control to the appropriate target in the code cache.

In the ARM ISA, any instruction can reference the PC as either a source or destination register. Pin's JIT compiler modifies any such instruction according to the following rules:

PC Read Instructions If an instruction references the PC as a source register, the JIT compiler replaces the instruction by a sequence. First, it sets up a constant (original_PC+8) in a temporary register. Next, it includes the original instruction with the PC reference replaced by a reference to the temporary register (Figure 2 (a)). Temporary registers are searched with liveness analysis. If no dead register is found, a live register is temporarily spilled around this code sequence to guarantee that the sequence can always be generated.

PC Write and Jump Instructions Unlike other ISAs, on ARM any instruction can target the PC as destination register. This effectively performs the operation of the instruction followed by a jump to the result of the operation. Our JIT compiler breaks up this instruction into a sequence. It first produces code to spill some context onto the VM stack, followed by code to compute the target address into a known register, and finally code to transfer control to the VM to find the target of the indirect jump in the code cache (Figure 2 (b)). A jump is a simple form of this rewriting where the operation itself is just a move (MOV PC ⇐ Rx).

The ARM ISA also allows base register updates for memory instructions. This is a side effect of the instruction and, on loads, effectively produces an additional result. If a load or load-multiple instruction targets the PC *and* has a base register update, the JIT compiler includes the base register in the spilled context. This is followed by the load instruction without the base register update, and then two more instructions to emulate the base register update

(a) PC Read Instruction

```
original code:  <OP> Rn ⇐ PC, ...

JIT compiled:   [spill Rtmp if necessary]
                MOV  Rtmp ⇐ #<original PC> + 8
                <OP> Rn ⇐ Rtmp, ...
                [fill Rtmp if necessary]
```

(b) PC Write Instruction

```
original code:  ADD PC ⇐ R0, R1 sll #2

JIT compiled:   STM SP! ⇐ {<context>}
                ADD Rtarget ⇐ R0, R1 sll #2
                B   <VM or indirect jump predictor>
```

(c) PC Write Load with Base Register Update

```
original code:  LDR PC ⇐ [R0], #4

JIT compiled:   STM SP! ⇐ {<context incl. R0>}
                LDR Rtarget ⇐ [<R0>]
                ADD Rtmp ⇐ <R0>, #4
                STR Rtmp ⇒ [SP, #<offset of R0>]
                B   <VM or indirect jump predictor>
```

(d) Call Instruction

```
original code:  BL <offset>

JIT compiled:   MOV LR ⇐ #<original PC> + 4
                B <VM or compiled target in CC>
```

Figure 2: Code samples showing JIT compiler transformations on PC references.

on the spilled version of the base register (Figure 2 (c)). When the VM returns to the application code after locating the target code in the code cache, it loads the spilled context, including the updated base register, back into the machine registers.

Call Instructions Direct branches and indirect jumps on ARM also exist in the form of calls that deposit the follow-on PC of the call (PC+4) into the LR register to be used by a later return instruction (`MOV PC ⇐ LR`). Pin's JIT compiler breaks up call type instructions into a separate update of LR followed by a regular branch or jump (Figure 2 (d)).

3.2 Direct and Indirect Branches

To ensure that the VM maintains control of execution at all times, and control never escapes back to the original, uninstrumented code, all branches within the cached code are patched and redirected to their transformed targets within the code cache. For those code regions that branch to code that is not yet present in the code cache, the system directs execution through an *exit stub* which performs two tasks. First, it stores the execution context of the running application (so that we may resume execution after building the next instrumented code region). Second, it transfers control back to the JIT compiler, passing along information about the next region to compile. Since traversing an exit stub and returning to the JIT is a very expensive task, the system attempts to avoid this step whenever possible, as we describe below.

Direct Branches In many instances, it is possible to avoid the overhead of traversing an exit stub, performing a state switch to VM mode, and entering the VM. A fairly straightforward instance is when an existing cached instruction sequence contains (or ends in) a direct branch. The moment that branch target appears in the code cache, Pin will patch any relevant direct branches to jump directly to that target trace. One alternative to our *pre-emptive linking* approach is to employ *lazy linking*, where we wait until control enters the VM and we discover that the intended target is already present in the code cache. At that time, we can patch the two traces together. We found that it wasn't worth the overhead of entering the VM unnecessarily, and that it was better to go ahead and link all potential branches as soon as the code is inserted, even at the risk of linking branches that may never be executed at run time.

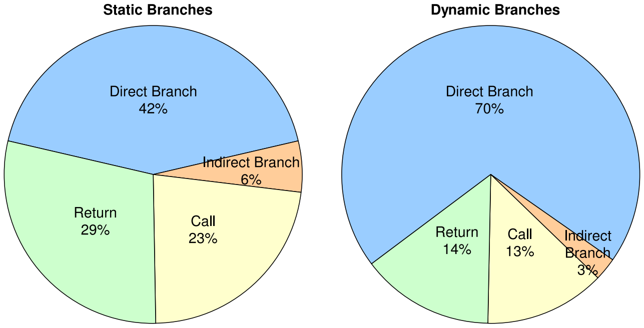


Figure 3: Static (executable) and dynamic (runtime) branch distribution for ARM. Results are averaged over the same subset of Spec 2k benchmarks as in Section 4.

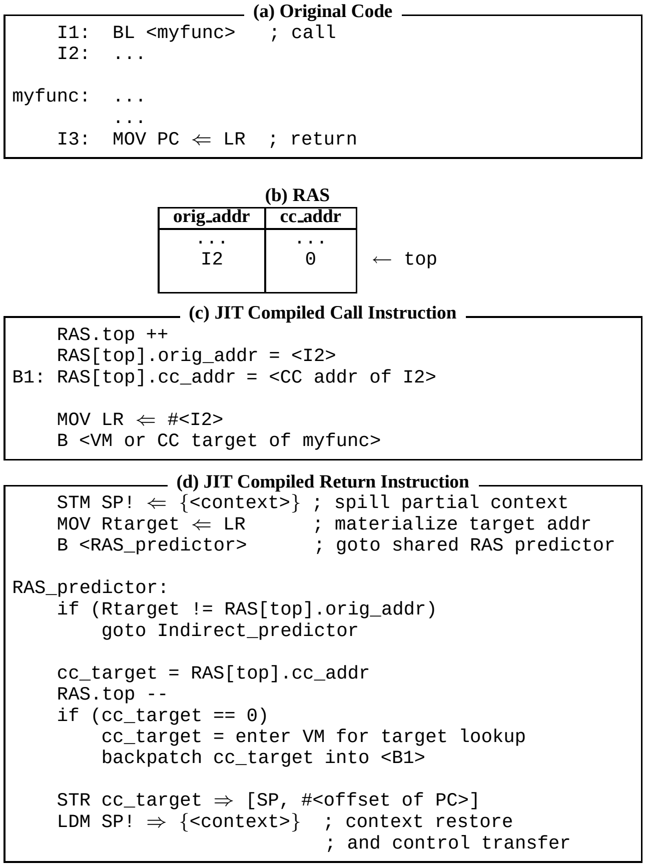
Indirect Branches A major challenge for any dynamic instrumentation system is how to handle indirect branches, which can result from `switch` statements in the source code, function returns, and indirect call stubs of shared library functions. These branches cannot be automatically patched to their intended target in the code cache, as the target varies at run time. Yet, as Figure 3 indicates, they are not uncommon.

The general solution is to transfer control to the VM which will look up the code cache location of the original target PC in its internal directory and will then transfer control to that code cache location. However, entering and leaving the VM as well as the directory search of the target inside the VM are very expensive operations. They represent an overhead of several orders of magnitude over the execution of the original instruction. To remedy this situation, Pin for ARM implements an indirect jump predictor. Instead of transferring control to the VM for target resolution, partial state is saved, a per-branch target prediction table pointer is loaded, and control is transferred to a tightly coded table lookup. If the original target is found in the table, the saved context is restored and control is transferred immediately to the corresponding code cache location (also stored in the table). If the target is not found in the local prediction table, the predictor performs a full (heavyweight) entry into the VM which engages in a full code cache directory search and potentially a JIT compile of the target if it is not yet present in the code cache. Finally, the local predictor table is updated with the target prediction so it can be found by the jump predictor the next time the same indirect jump is executed.

The jump predictor works well for indirect branches that have a small number of different targets. It does not perform well for returns from functions that have many different call sites, therefore we also implemented a return address stack.

Return Prediction with a Return Address Stack As can be seen in the overall branch profile Figure 3, return instructions are the most common form of indirect branches executed. In order to predict returns efficiently, we have implemented a software Return Address Stack [22] in Pin for ARM.

Figure 4 (a) depicts the original application code of a call `I1` and an associated return `I3` of the called function `myfunc`. The return will transfer control back to `I2`, the instruction after the call. If



(a) Original Code

```
    I1:  BL <myfunc>   ; call
    I2:  ...

myfunc:  ...
         ...
    I3:  MOV PC ⇐ LR  ; return
```

(b) RAS

orig_addr	cc_addr	
...	...	
I2	0	← top

(c) JIT Compiled Call Instruction

```
    RAS.top ++
    RAS[top].orig_addr = <I2>
B1: RAS[top].cc_addr = <CC addr of I2>

    MOV LR ⇐ #<I2>
    B <VM or CC target of myfunc>
```

(d) JIT Compiled Return Instruction

```
    STM SP! ⇐ {<context>} ; spill partial context
    MOV Rtarget ⇐ LR      ; materialize target addr
    B <RAS_predictor>      ; goto shared RAS predictor

RAS_predictor:
    if (Rtarget != RAS[top].orig_addr)
        goto Indirect_predictor

    cc_target = RAS[top].cc_addr
    RAS.top --
    if (cc_target == 0)
        cc_target = enter VM for target lookup
        backpatch cc_target into <B1>

    STR cc_target ⇒ [SP, #<offset of PC>]
    LDM SP! ⇒ {<context>}  ; context restore
                           ; and control transfer
```

Figure 4: Predicting return instructions with a Return Address Stack (RAS) predictor.

return prediction is enabled, the VM also keeps an auxiliary data structure in its private memory, the Return Address Stack (RAS) as depicted in Figure 4 (b).

When the JIT compiler translates the call instruction `I1`, it produces the code shown in Figure 4 (c) and puts it into the code cache. The code first pushes the original return address `I2` in the $orig\_addr$ field of a new location on the RAS. It then stores the code cache address of the translation of `I2` into the cc_addr field of the same RAS entry. If `I2` is not yet in the code cache, a 0 value is written into the cc_addr field. In addition, the code cache is instructed to store the address of the generated store instruction `B1` as an unresolved incoming edge to target `I2`. Finally the LR update and control transfer instructions for the original call are generated.

Let's assume the call has executed and has pushed its `<I2,0>` entry onto the RAS and execution now reaches the return instruction which has been converted into a call to the RAS predictor lookup depicted in Figure 4 (d). If the original target PC `Rtarget` is found on top of the RAS, the cc_target is loaded and the RAS is popped. Since the cc_target of the `<I2,0>` RAS entry is unknown (0), the VM is entered and the JIT compiles the code for `I2`. When the compiled code is entered into the code cache, the previously recorded unresolved incoming edge to target `I2` is found. This initiates a back-patch of the source of the incoming edge, instruction `B1`, overwriting the previous placeholder value of 0 with cc_target, the real code cache address of instruction `I2`. Finally execution is resumed at cc_target.

The next time the call `I1` executes, it pushes the correct entry `<I2,cc_I2>` onto the RAS and the associated return will find the code cache target of `I2` on the RAS, without the need to look it up in the VM. Thus, the return is predicted correctly.

An interesting challenge on ARM is that it is not always clear whether an instruction is a return instruction or an arbitrary indirect jump. The ARM ISA does not feature a special return instruction. Instead, any instruction that targets the PC as destination register can potentially be used as return. The simple solution that Pin for ARM employs is to always try to predict an indirect jump first with the RAS. If the target matches the $orig\_addr$ on top of the RAS, the instruction was a return instruction and will pop and update the RAS. If the top of the RAS does not match, the RAS is not updated and a table based indirect-branch prediction is attempted. Only if that fails do we have to pay the heavy price of a VM entry and full code cache directory search. Note that since the RAS is really implemented as a small circular buffer of limited size instead of a true stack, any potential errors in correlating RAS pushes with pops are *self healing* in that they merely degrade predictor performance somewhat but do not cause functional errors.

3.3 System Calls

From an ISA standpoint, system calls do not present any particular problem in Pin for ARM, since they can be executed directly without further intervention from Pin. However, in order to stay in control of the application under all circumstances, some system calls must be intercepted and emulated instead. These system calls include the $signal$ and $thread$ related system calls in order to be able to intercept the delivery of signals and creation of additional application threads. Pin on IA32 already has full support for signals and multi-threading. Similar support is currently under development in Pin for ARM as well.

Pin also monitors `mmap` related system calls to manage the location of mapped regions between the application, the Pin VM, and the Pin tool, all of which share the same address space. Any `munmap` calls must be tracked by the code cache, to ensure that the cached code remains consistent with the original application and shared libraries. Finally, the `exit` system call is monitored in order to cleanly shut down Pin, e.g. writing out final statistical reports, rather than having the operating system terminate Pin involuntarily upon application exit. Note that the number of intercepted and emulated system calls is typically very small (less than 100 for all our benchmarks) such that we have not found system call handling techniques to be performance critical.

3.4 Code Caches and Policies

As mentioned earlier, Pin uses a code cache to store previously instrumented copies of the application to amortize its overhead. If not carefully managed, the code cache and its associated metadata can quickly consume massive amounts of memory [20]. While this is simply a performance issue on some systems, it becomes a correctness problem when one considers the limited memory available on embedded systems.

Pin's code cache is partitioned into multiple equal-sized *cache blocks* in memory that are generated on demand. This configuration allows the code cache to adapt to increasing application requirements. In the default case, each cache block is sized at `(PageSize * 16)`, which evaluates to 64 kB on IA32, EM64T and ARM, and 256 kB on IPF. Pin's entire code cache is unbounded by default on IA32, EM64T and IPF, while a 16 MB limit is placed on the ARM code cache due to hard limitations on resources. Users may override or dynamically adjust the code cache and cache block sizes at run time using a code cache client API [18].

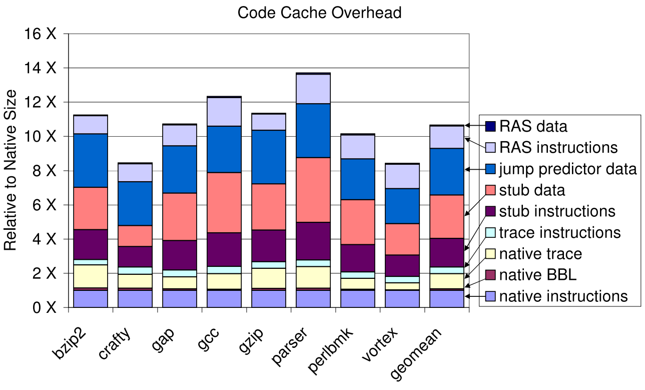


Figure 5: Max unbounded code cache size for our benchmarks.

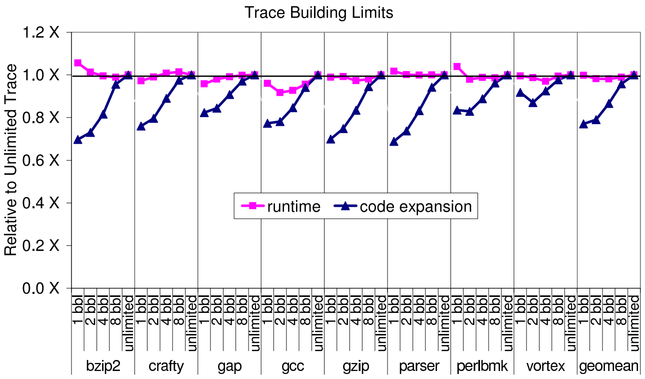


Figure 6: The effects of limiting the trace size on code cache size and run-time performance.

The code cache stores application traces, exit stubs, and data. One exit stub is required for every application branch appearing in the code cache, and one data region is required for every exit stub (to store relevant application state). Additional stubs and data are used for performance features such as indirect branch prediction.

While many systems assume that the majority of the code cache consists of application traces, the reality is quite different [17]. Figure 5 shows a breakdown of the code cache contents by category. This breakdown turns out to be much more dominated by exit stubs and data for ARM than for many other architectures.

We take as baseline *native instructions*, the cumulative size of all instructions executed during the run of the program. Pin only translates executed instructions, thus we do not count unexecuted instructions from the binary image. The next two categories *native BBL* and *native trace* extend the coverage to whole basic blocks and traces respectively, accounting for the overhead of building larger fetch regions (see Section 3.5). Now entering the realm of genuine Pin overhead, *trace instructions* adds the overhead of breaking up some instructions into longer sequences, such as those that reference the PC (see Sec. 3.1). The main tasks of exit stubs are to save the application context onto the VM stack (2 instructions), to set up an argument pointer for the VM (1 instruction), and to jump to the VM (1 instruction). Each stub contributes to $4 * 4 = 16$ bytes of *stub instruction* overhead and $2 * 4 = 8$ bytes of *stub data* overhead as well as roughly 44 bytes for the VM argument structure. When enabling the indirect jump predictor, an 8-entry target table (*jump predictor data*) is also kept in the code cache for every indirect branch. Another significant contributor to code expansion is RAS-based return prediction. Each call site includes code (19 instructions) to push two 32-bit constant values onto the revolving RAS, contributing to 76 bytes of *RAS instruction* overhead and an additional 4 bytes of *RAS data* overhead.

Since the net result from this analysis is that a code expansion of over 10x is typical, limiting and managing code cache size is quite important on ARM. While on some architectures it is worthwhile to perform fine-grained code cache evictions [19], this result does not hold for embedded architectures where memory is one of the most critical resources. Therefore, we implement a simple, yet efficient flush-on-full policy in Pin for ARM.

3.5 Trace Selection

Superblocks (single-entry, multiple-exit regions) are used as the basis for instrumentation and code caching in Pin. Just before the first execution of a basic block, Pin speculatively creates a straight-line trace of instructions that is terminated by either (1) an unconditional branch, or (2) an instruction count limit. Pin's approach to trace selection, unlike other profile-based or statistical [14] schemes, stems from the fact that it is designed to be an instrumentation system, thus it is important for traces to reside in contiguous memory.

One ARM-specific trace selection optimization we explored was to limit trace lengths to a fixed maximum number of basic blocks. This optimization reduces the *tail duplication* resulting from caching superblocks (since side entries are not permitted, occasionally one superblock may be duplicated within another). Figure 6 shows the results of this optimization for traces of 1, 2, 4, and 8 basic blocks as well as unlimited basic blocks per trace. The figure shows the run time and code expansion relative to that obtained by unlimited basic blocks per trace (there is a limit of 70 instructions per trace, however). We see that limiting trace sizes does reduce the code cache footprint, though that reduction will only translate into notable run-time performance improvements if we reduce the size to the point where it will fit in the hardware cache. Sometimes, as in the case of `bzip2`, smaller traces actually degrade performance. In these cases, the benefits of smaller footprints are outweighed by an increase in the number of context switches into trace generation mode. For the remainder of this paper, all results were obtained by using traces with an unlimited number of basic blocks.

3.6 Self-Modifying Code

A major challenge in many dynamic instrumentation systems is self-modifying code (SMC). Any time an application modifies its own code region, the instrumentation system must be aware of this change in order to invalidate, regenerate, and re-instrument its cached copy of the modified code. Otherwise, the system will continue to execute the stale, cached copy. Self-modifying code is generated, for example, by managed runtime systems like Java, therefore it should be handled in a robust instrumentation framework.

Detection The challenge turns out not to be efficient *handling* of self-modifying code, but efficient *detection*. On IA32, for example, SMC detection requires expensive page-protection mechanisms to mark the original code as unwritable, then to catch and handle any write attempts to code regions. This becomes particularly challenging in the face of mixed code and data (prevalent on IA32), which triggers a great deal of SMC false alarms. Fortunately, architectures such as ARM and IPF all but trivialize SMC detection, as the architecture contains an explicit instruction that must be used by the software developer in order to correctly implement SMC.

Handling In Pin for ARM (and for IPF), we simply watch the instruction stream for this special instruction, and handle the self-modifying code using the same techniques as on other architec-

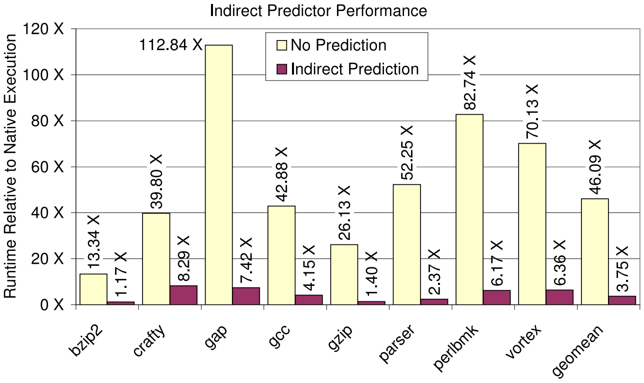


Figure 7: Performance improvements due to indirect branch prediction.

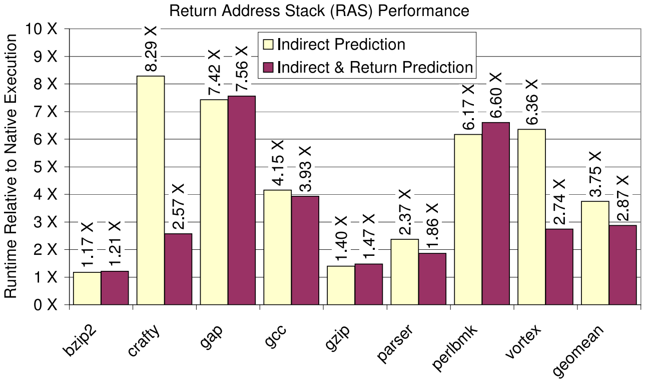


Figure 8: Performance improvements due to the Return Address Stack (RAS).

tures. This instruction on ARM (write register in System Control Co-processor) is protected. Since Pin is operating at the user level, what we actually have to watch is the interface provided by the operating system to invoke this functionality, which is the ARM specific Linux system call `__ARM_NR_cacheflush`. We simply invalidate the corresponding cached code, knowing that the next time that code is executed, it will be regenerated and cached.

4. PERFORMANCE ANALYSIS

It goes without saying that performance overheads are important to the user experience. While users are willing to tolerate a reasonable amount of overhead if the benefits are high enough, it is an uphill battle to maintain users when the system degrades performance by an order of magnitude or more. As such, we've put a great deal of effort into driving the performance overheads of Pin for ARM down to their current levels.

Our results were gathered on an iPAQ PocketPC H3835 running Intimate Linux with kernel 2.4.19. The iPAQ has a 200 MHz StrongARM-1110 processor and 64 MB RAM. We report results relative to the native performance of each benchmark (without Pin) on the same hardware and software configuration.

We use a subset of the Spec 2k benchmarks for our performance studies. Although Spec is not a widely applied benchmarking set in the embedded processor world, we are accustomed to using it across Pin's supported architectures (IA32, EM64T, IPF, ARM) in order to compare Pin's performance across different architectures. In order to be able to run on the restricted resources of our test machine, we have reduced the input data set of the benchmarks to values that result in native run times of approximately 1-15 minutes. We also limit the benchmark data set sizes to fit within approximately 1/2 the available physical memory on the machine and have turned off paging. Since our test hardware does not contain a hardware floating point unit, we have only selected benchmarks that do not contain a significant number or FP operations.

4.1 Indirect Prediction Performance

As described earlier in Section 3.2, Pin for ARM spends a lot of effort predicting the code cache target of indirect jumps rather than going through a full VM entry and code cache directory search.

Indirect Branch Prediction Figure 7 shows a performance comparison of Pin for ARM without and with indirect branch prediction. As we can see, without indirect branch prediction the benchmark run times are on average between one and two orders of magnitude worse than the native application. In the worst case we see a slowdown of over 112 X. This overhead is clearly not acceptable to users which would like to use Pin for ARM for application studies. Adding indirect branch prediction with a per branch prediction table of 8 targets brings the average performance loss down to approximately 4 X slower than native, over an order of magnitude better than without the predictor. While some benchmarks like `crafty` and `gap` still show a significant slowdown of close to an order of magnitude, others like `bzip2` and `gzip` run at close to native speed. The most impressive performance improvement is achieved by `parser` which executes more than 20 times faster with indirect code cache target prediction under Pin for ARM.

Return Prediction From the data in Figure 7 we saw the importance of indirect branch prediction in general. Figure 8 focuses on the performance effects of return address prediction with the RAS as described in Section 3.2. Starting with an average slowdown of 3.75 X, adding a return predictor brings the average performance loss down to 2.87 X. It is particularly effective for `crafty` and `vortex` where it reduces the performance loss to much less than half. Due to the added overhead of maintaining a RAS (push and pop of return address tuples at every call and return respectively) some benchmarks like `gap` and `perlbmk` suffer minor performance losses. Although these benchmarks do have considerable call/return traffic, each return only jumps to a small number of call sites. In such cases the generic indirect jump predictor can capture all return targets in its 8-entry prediction table and the added overhead of keeping a RAS outweighs its benefits. Overall, however, adding a RAS has improved performance considerably.

4.2 Remaining Prediction Problems

As can be seen in Figure 8, `gap` and `perlbmk` still suffer high overheads. Our analysis has revealed that these overheads are caused by indirect prediction misses. An indirect prediction miss costs us approximately 45 μs (about 9000 clock cycles) of processing time in the VM code cache directory lookup. Using this knowledge we arrive at Figure 9. It gives a breakdown of the relative time spent in the native execution of the applications, the overhead of Pin if it could predict indirect jumps perfectly, and the additional overhead of the indirect prediction misses. Both `gap` and `perlbmk` show an overhead of 4-5 X which could be removed with a better indirect prediction scheme. We are still investigating ways to reduce the number of prediction misses further within the constraints of adding additional space, search, and maintenance time overhead for possible candidate solutions.

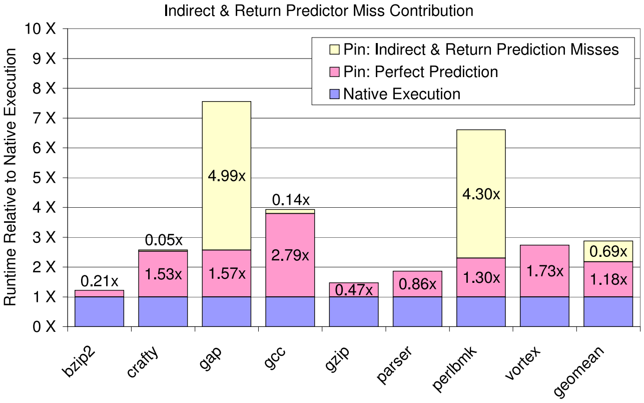


Figure 9: Remaining contribution of indirect and return predictor misses.

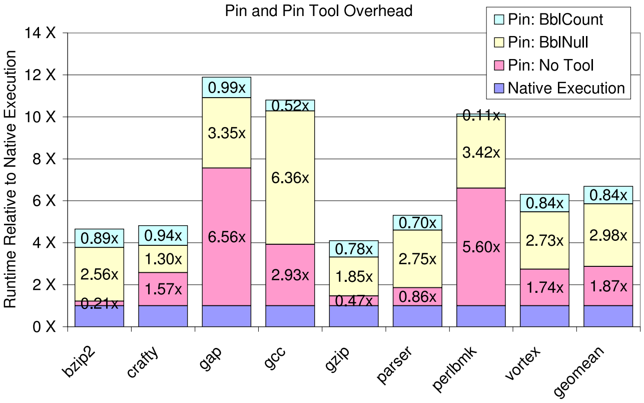


Figure 10: Performance of PinARM with and without Pintools relative to native.

4.3 Overall Pin and Pin Tool Overhead

We report the overall results of our efforts in Figure 10. For each benchmark, we present the following overhead:

- Benchmarks with Pin - This test illustrates the baseline overhead of Pin without employing any instrumentation.
- Benchmarks with Pin and BblNull Tool - This test exercises many of Pin's APIs and measures their overhead. The end result is that Pin will embed empty instrumentation calls into the executable.
- Benchmarks with Pin and BblCount Tool (source code shown in Figure 11) - This test illustrates the overhead that a user would see if they wished to use Pin to count the number of instructions executed at runtime. It has the same *instrumentation* overhead as BblNull, but adds *analysis* overhead. In each call to the analysis routine, the number of executed instructions in the basic block is added to a global 64-bit counter. Figure 11 (b) shows that this contributes seven instructions of additional overhead per analysis call.

The results in each category in Figure 10 represent the additional overhead contributed by that category relative to native execution. For example, on average we see that running the benchmarks under Pin for ARM contributes 1.87 X of overhead. Adding an empty analysis function (BblNull) at the entry of each basic block adds another 2.98 X of overhead. Finally, adding contents to the analysis functions (BblCount) contributes 0.84 X of overhead.

(a) BblCount Pin Tool Code

```
#include <iostream>
#include "pin.H"

UINT64 icount = 0;

VOID DoCount(INT32 c)
{
  icount += c;
}

VOID Trace(TRACE trace, VOID *v)
{
  for (BBL bbl = TRACE_BblHead(trace);
       BBL_Valid(bbl);
       bbl = BBL_Next(bbl))
  {
    INS_InsertCall(BBL_InsHead(bbl),
       IPOINT_BEFORE, (AFUNPTR)DoCount,
       IARG_UINT32, BBL_NumIns(bbl),
       IARG_END);
  }
}

VOID Fini(INT32 code, VOID *v)
{
  std::cerr << "Count: " << icount << endl;
}

int main(INT32 argc, CHAR **argv)
{
  PIN_Init(argc, argv);

  TRACE_AddInstrumentFunction(Trace, 0);
  PIN_AddFiniFunction(Fini, 0);

  PIN_StartProgram();
  return 0;
}
```

(b) DoCount Disassembled

```
DoCount:
  ldr     ip, [pc, #24]       ; & icount
  str     r4, [sp, -#4]!      ; spill
  ldmia   ip, {r3, r4}        ; load icount
  adds    r1, r3, r0          ; icount += c (low)
  adc     r2, r4, r0, asr #31 ; icount += c (high)
  stmia   ip, {r1, r2}        ; store icount
  ldmia   sp!, {r4}           ; fill
  mov     pc, lr              ; return
```

Figure 11: Source code for the `BblCount` Pintool.

Various factors contribute to the diverse distributions of these factors across the different benchmarks. `Gcc` for example sees the largest overhead of 6.36 X for adding the BblNull instrumentation. This is due to the fact that `gcc` has the largest native code footprint of 533 kB of all benchmarks analyzed. Due to the large number of different basic blocks instrumented, Pin spends a proportionally larger amount of time in the JIT compiler generating the instrumentation stubs. Also, due to the large size of the generated code of more than 6.5 MB, the small instruction cache of 32 kB of the StrongARM processor becomes less effective as it can not capture the working set efficiently. On the other hand we see that the overhead of BblCount for `gcc` is relatively small at 0.52 X. This is because basic blocks in `gcc` are sufficiently large such that the additional execution of seven more assembly instructions per basic block does not incur much overhead.

On the other hand, `gzip`, which has the smallest working set size of only 27 kB, exhibits a small slowdown of only 1.85 X for adding the null instrumentation due to the small number of instrumentation stubs generated by the JIT compiler and the resulting smaller generated code size of 312 kB.

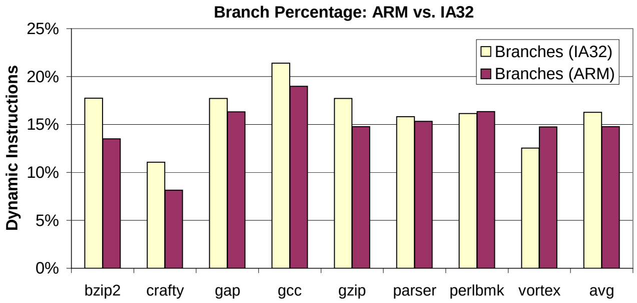


Figure 12: Architectural comparison of branch vs. non-branch instructions executed at run time.

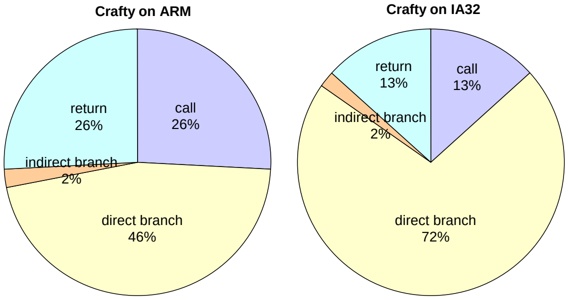


Figure 13: Run-time branch distribution for `crafty`.

5. POTENTIAL APPLICATIONS

Just as program instrumentation opened the doors for many research opportunities within the high-performance computing domain, the same (if not more) potential is present in the embedded domain. The added challenges of an embedded environment truly test the limits of many algorithms and implementation decisions. Designers can use dynamic instrumentation to rapidly prototype new ideas, debug existing bottlenecks, and even add new functionality to existing hardware and software. These applications are just as relevant in the context of embedded systems as they are in high-performance general-purpose computers. In the following sections, we will discuss several potential applications of Pin and other instrumentation systems in more detail.

5.1 Software Introspection

Understanding software applications is the key to effective hardware design. Designing for the common case requires an understanding of the common case, and this goal is best accomplished by measuring various attributes of new and existing applications. Instrumentation tools like Pin greatly simplify the measurement process, allowing users to inspect the state of the system after every executed instruction. While similar details could be acquired by manually modifying millions of lines of code over numerous applications and shared libraries, Pin allows its user to write a single plug-in (Pintool) that gathers the desired information in rarely more than 100 lines of code. This single Pintool can then be applied to countless applications and libraries, regardless of whether or not the application source code is still (or was ever) available.

There are numerous application features that can be directly measured using Pin, including details about static and dynamic instruction frequencies, branch behavior, and memory usage. Writing a Pintool that outputs the overall dynamic instruction count can be accomplished in approximately 15 lines of C code using Pin's API (and indeed this Pintool – called `icount` – is shipped with Pin.) Pin is currently packaged with several dozen open-source plug-in tools that perform actions such as instruction tracing, memory tracing, cache simulation, a range of statistical code analyzers, call graph analysis, memory allocation/deallocation checks, code coverage testing, etc. In fact, most of the results in Sections 3–4 were gathered using existing Pintools that are shipped with Pin. These tools may be used directly, or may be used as a template for developing other tools. (More details and features of various Pintools are available in the Pin manual and on the Pin web site [21].)

Pin was designed to meet the need for software introspection, thus its contribution to the embedded community is just as significant as it is to the high-performance community, as understanding application behavior is equally important to both communities.

5.2 Architecture and Compiler Design

As we mentioned earlier, the Pin dynamic instrumentation system is unique in that it currently supports four architectures: IA32, EM64T, IPF, and ARM. This feature gives users the ability to perform comparisons of application behavior across different architectures. A user can develop a simple, system-independent Pintool, then compile and execute it on all four architectures. While we might expect the same application to behave similarly on different architectures, Figures 12–13 show that results often vary widely between architectures.

Figure 12 shows that the percentage of branches on IA32 is not a good predictor of the percentage of branches on ARM even though the same applications and input sets (`test`) were used. Both the architecture and the compiler[2] changed, and for applications like `bzip2`, this resulted in 22% fewer executed branches on ARM than on IA32. The variation results from ARM's support for (and the compiler's use of) predication, which alters both the number of generated branches and the behavior of those branches. Therefore, it's clear that embedded systems designers should design branch predictors using results gathered by an embedded instrumentation system like Pin for ARM, because the IA32 branch results do not necessarily correlate.

More evidence is shown in Figure 13 where we see the distribution of branches in the `crafty` benchmark. Clearly, the ARM version of `crafty` exhibits a much higher percentage of call-return sequences than the IA32 version. The reason is quite intuitive, since ARM supports predication, thus many of the direct branches have been elided. This result makes it clear that the designers of ARM systems should focus on indirect branch performance, as we have in Pin, because a great performance benefit may result.

Even more evidence of an architectural variation can be seen in Table 1 where we used the `opcodemix` Pintool to output the dynamic frequency of the top five opcodes used by `bzip2`. We see that explicit `load` instructions are much more common on ARM, as almost any instruction can perform a `load/store` on IA32. Also, explicit `branch` instructions are much more common on IA32 since ARM is predicated and any instruction can write to the PC, effectively performing a jump. These features will undoubtedly impact hardware and software design of embedded systems.

Instrumentation systems also enable compiler developers to understand, debug, and improve code generation techniques. For instance, Pin has been used to improve the quality of code generated by a proprietary ARM compiler tool chain. First, the compiler developer compiles an application using two different compilers (different tool chains, compiler versions, or even two different target

[2] We used the `arm-linux-gcc` cross compiler for ARM and the `gcc` native compiler for IA32.

Rank	Opcode (ARM)	Dynamic Frequency	Opcode (IA32)	Dynamic Frequency
1	add	16.99%	movzx	24.76%
2	cmp	16.61%	mov	19.65%
3	ldrb	15.82%	cmp	15.88%
4	mov	15.34%	inc	13.14%
5	ldr	12.44%	jnz	10.73%

Table 1: Top-5 opcodes executed while running bzip2 on two different architectures. The second and fourth columns report the dynamic execution frequency of the corresponding opcode.

architectures). Then, those two binaries are executed using Pin and a Pintool similar to opcodemix, which outputs static and dynamic instruction frequencies. A post-processing comparative analysis tool can then present statistics at the global and per function level. Finally, these statistics are used to pinpoint inefficiencies in the generated code, e.g. due to inefficient code templates or idioms being used in the code generation phase.

5.3 Architectural Compatibility

The ARM architecture is regularly extended and refined. Within the past five years, we have seen a transition from the ARM ISA version 5 to version 6. The main improvements in version 6 include improved memory management, multiprocessing support, multimedia support and improved data handling, while still providing full support for backward compatibility with version 5.

Yet, as one might expect, it is infeasible to provide *forward compatibility* such that any future ARM binary can execute on existing hardware. Newly introduced instructions will result in an illegal instruction exception when executed on older hardware. One interesting application of Pin for ARM is that it can be used to detect and emulate any instructions from future ISA versions. The microprocessors in our iPAQs implement an older version of the ARM ISA (version 4). We have used this emulation technique to execute newer binaries on this older hardware. The same technique has been used to evaluate the effects of proposed new ISA extensions before they are actually implemented on any existing hardware.

Not all embedded applications were designed to run on a real operating system as they do on our Linux-based iPAQ machines. In the embedded world, it is customary to run applications on a very rudimentary OS or even the bare hardware. We have taken such *baremetal* applications and have successfully executed them under Pin on Linux by having the JIT compiler generate calls to OS emulation code for all SWI instructions (system calls) requesting OS services. This allowed us to run any Pin tool on these baremetal applications in order to do performance studies on them, which would not be possible directly on a baremetal system.

5.4 Instrumentation System Design

As can be inferred from this paper, the design of a dynamic binary instrumentation system consists of countless intermediate design decisions. This makes instrumentation system design a very challenging, yet interesting research area. To enable outside research into this design space, several API hooks have been integrated into Pin that provide user control to the internal workings and behavior of Pin itself. This allows a novice user to experiment with design decisions in dynamic instrumentation systems for various architectures and operating systems, without the need to access and understand a large amount of source code. Furthermore, several specialized sets of Pin APIs have been announced, including the optimization and trace generation interface [31] and the code cache design interface [18].

6. RELATED WORK

Dynamic instrumentation for high-performance architectures is well-charted territory [3, 23, 29]. Furthermore, a similar internal design is frequently used in dynamic optimizers [6], dynamic translators [4, 7], architecture simulation tools [8, 25], profilers [26], and co-designed virtual machines [1, 12, 15].

Embedded systems differ in many ways from high-performance systems, thus binary translators for embedded systems are quite different in character. Perhaps the closest related work to ours is the DELI [13] system from Hewlett-Packard and ST Microelectronics, which translates and optimizes code for the LX embedded architecture. The main distinctions between DELI and Pin for ARM are the end goals (optimizations vs. instrumentation), the target architectures (LX vs. ARM), and their availability to the community.

Java virtual machines [9, 10, 11, 28, 30] are widely used at the embedded level. Because JVMs do a form of translation on-the-fly, they serve a similar purpose as dynamic translation systems, however the internal structure of most JVMs is quite different.

There are several techniques that have been used for embedded system virtual platforms to make them lightweight and feasible to implement [2, 16, 24, 28, 30]. Some approaches even use the embedded system as a client with a more powerful server [27, 32]. The server is used to compile, optimize, and store code using either online or offline profiles as a guide for optimization and or storage. These approaches are quite dissimilar to the unified approach implemented in Pin.

7. CONCLUSIONS

Dynamic instrumentation systems have proven to be quite useful to software developers and end users. Until now, efficient, robust, easy-to-use instrumentation tools for embedded systems haven't been readily available. In this paper, we presented the overall design, implementation, performance, and applications of the ARM version of Pin. We detailed the design decisions we made that were geared toward the idiosyncracies of the ARM architecture, and describe how we handle challenges such as indirect branches and self-modifying code. Our efforts improved the performance of the overall system to the point where it rivals the performance of the IA32 implementation, despite the unique challenges and constraints of embedded systems.

Dynamic instrumentation systems have opened the doors for a multitude of research directions, often many more directions than the original system designers had ever anticipated. While we have initially intended Pin for ARM to be used for program instrumentation, it could also be used to explore opportunities for security, reliability, program adaptation, and optimization on embedded architectures.

Acknowledgments

The Pin project is a collaborative effort supported by Intel Corporation and developed over several years by a large team of researchers, including Geoff Lowney, Robert Cohn, Robert Muth, Greg Lueck, C.K. Luk, Steven Wallace, Harish Patil, Mark Charney, Vijay Janapa Reddi, and Ramesh Peri. We also thank the anonymous reviewers for their constructive feedback.

8. REFERENCES

[1] E. R. Altman, M. Gschwind, S. Sathaye, S. Kosonocky, A. Bright, J. Fritz, P. Ledak, D. Appenzeller, C. Agricola, and Z. Filan. BOA: The architecture of a binary translation processor. *IBM Research Report RC 21665*, Dec 2000.

[2] D. F. Bacon, P. Cheng, and D. Grove. Garbage collection for embedded systems. In *4th ACM International Symposium on Embedded Software*, pages 125–136, Sep 2004.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *ACM Conference on Programming Language Design and Implementation*, pages 1–12, Jun 2000.

[4] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium-based systems. In *36th Intl. Symp. on Microarchitecture*, pages 191–201, Dec 2003.

[5] D. Bruening and S. Amarasinghe. Maintaining consistency and bounding capacity of software code caches. In *3rd Intl. Symp. on Code Generation and Optimization*, Mar 2005.

[6] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *First Intl. Symp. on Code Generation and Optimization*, pages 265–275, Mar 2003.

[7] C. Cifuentes, B. Lewis, and D. Ung. Walkabout - a retargetable dynamic binary translation framework. Technical Report TR2002-106, Sun Microsystems Laboratories, Jan 2002.

[8] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994.

[9] M. Debbabi, A. Gherbi, L. Ketari, C. Talhi, N. Tawbi, H. Yahyaoui, and S. Zhioua. A dynamic compiler for embedded java virtual machines. In *3rd International Symposium on Principles and Practice of Programming in Java*, pages 100–106, Jun 2004.

[10] M. Debbabi, A. Gherbi, L. Ketari, C. Talhi, H. Yahyaoui, and S. Zhioua. A synergy between efficient interpretation and fast selective dynamic compilation for the acceleration of embedded java virtual machines. In *3rd International Symposium on Principles and Practice of Programming in Java*, pages 107–113, Jun 2004.

[11] M. Debbabi, A. Mourad, and N. Tawbi. Armed e-bunny: a selective dynamic compiler for embedded java virtual machine targeting arm processors. In *ACM Symposium on Applied Computing*, pages 874–878, Mar 2005.

[12] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *First Intl. Symp. on Code Generation and Optimization*, pages 15–24, Mar 2003.

[13] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. Deli: A new run-time control point. In *35th Intl. Symp. on Microarchitecture*, pages 257–268, 2002.

[14] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 202–211, Oct 2000.

[15] K. Ebcioglu and E. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *24th Intl. Symp. on Computer Architecture*, pages 26–37, Jun 1997.

[16] P. Griffin, W. Srisa-an, and J. M. Chang. An energy efficient garbage collector for java embedded devices. In *Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 230–238, 2005.

[17] A. Guha, K. Hazelwood, and M. L. Soffa. Reducing exit stub memory consumption in code caches. In *High Performance Embedded Architectures and Compilers*, Jan 2007.

[18] K. Hazelwood and R. Cohn. A cross-architectural interface for code cache manipulation. In *6th Intl. Symp. on Code Generation and Optimization*, pages 17–27, New York, NY, Mar 2006.

[19] K. Hazelwood and J. E. Smith. Exploring code cache eviction granularities in dynamic optimization systems. In *2nd Intl. Symp. on Code Generation and Optimization*, pages 89–99, Palo Alto, CA, Mar 2004.

[20] K. Hazelwood and M. D. Smith. Generational cache management of code traces in dynamic optimization systems. In *36th Intl. Symp. on Microarchitecture*, pages 169–179, San Diego, CA, Dec 2003.

[21] Intel. Pin web pages. http://rogue.colorado.edu/Pin.

[22] D. R. Kaeli and P. G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *18th Intl. Symp. on Computer Architecture*, pages 34–42, 1991.

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapareddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, Jun 2005.

[24] M. Mamidipaka and N. Dutt. On-chip stack based memory organization for low power embedded architectures. In *Conference on Design, Automation and Test in Europe*, pages 1082–1087, 2003.

[25] M. Moudgill, J.-D. Wellman, and J. H. Moreno. Environment for powerpc microarchitecture exploration. *IEEE Micro*, 19(3):15–25, 1999.

[26] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[27] J. Palm, H. Lee, A. Diwan, and J. E. B. Moss. When to use a compilation service? In *Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 194–203, 2002.

[28] U. P. Schultz, K. Burgaard, F. G. Christensen, and J. L. Knudsen. Compiling java for low-end embedded systems. In *Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 42–50, 2003.

[29] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. L. Soffa. Reconfigurable and retargetable software dynamic translation. In *First Intl. Symp. on Code Generation and Optimization*, pages 36–47, Mar 2003.

[30] N. Shaylor, D. N. Simon, and W. R. Bush. A java virtual machine architecture for very small devices. In *Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 34–41, Jun 2003.

[31] Q. Wu, V. J. Reddi, Y. Wu, D. Connors, D. Brooks, M. Martonosi, and D. W. Clark. A dynamic compilation framework for controlling microprocessor energy and performance. In *38th Intl. Symp. on Microarchitecture*, pages 271–282, Nov 2005.

[32] S. Zhou, B. R. Childers, and M. L. Soffa. Planning for code buffer management in distributed virtual execution environments. In *1st ACM/USENIX International Conference on Virtual Execution Environments*, pages 100–109, 2005.