A Retargetable Static Binary Translator for the ARM Architecture

BOR-YEH SHEN, WEI-CHUNG HSU, and WUU YANG, National Chiao-Tung University

Machines designed with new but incompatible Instruction Set Architecture (ISA) may lack proper applications. Binary translation can address this incompatibility by migrating applications from one legacy ISA to a new one, although binary translation has problems such as code discovery for variable-length ISA and code location issues for handling indirect branches. Dynamic Binary Translation (DBT) has been widely adopted for migrating applications since it avoids those problems. Static Binary Translation (SBT) is a less general solution and has not been actively researched. However, SBT performs more aggressive optimizations, which could yield more compact code and better code quality. Applications translated by SBT can consume less memory, processor cycles, and power than DBT and can be started more quickly. These advantages are even more critical for embedded systems than for general systems.

In this article, we designed and implemented a new SBT tool, called LLBT, which translates ARM instructions into LLVM IRs and then retargets the LLVM IRs to various ISAs, including ×86, ×86–64, ARM, and MIPS. LLBT leverages two important functionalities from LLVM: comprehensive optimizations and retargetability. More importantly, LLBT solves the code discovery problem for ARM/Thumb binaries without resorting to interpretation. LLBT also effectively reduced the size of the address mapping table, making SBT a viable solution for embedded systems. Our experiments based on the EEMBC benchmark suite show that the LLBT-generated code can run more than 6× and 2.3× faster on average than emulation with QEMU and HQEMU, respectively.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation, Optimization, Retargetable compilers*

General Terms: Design, Languages, Performance

Additional Key Words and Phrases: Binary translation, compiler, retargeting, intermediate representation

ACM Reference Format:
Bor-Yeh Shen, Wei-Chung Hsu, and Wuu Yang. 2014. A retargetable static binary translator for the ARM architecture. ACM Trans. Architec. Code Optim. 11, 2, Article 18 (June 2014), 25 pages.
DOI: http://dx.doi.org/10.1145/2629335

1. INTRODUCTION

Binary Translation (BT) techniques have been studied and developed in the past two decades [Sites et al. 1993; Andrews and Sand 1992]. They have been widely adopted in many different areas, such as fast simulation, software security enforcement, application profiling [Luk et al. 2005], and system virtual machine implementations

---

This article extends the authors' previous work entitled "LLBT: An LLVM-based Static Binary Translator" published in the Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'12) in 2012.

The work reported in this paper is partially supported by National Science Council (NSC), Taiwan, Republic of China, under grants NSC 100-2218-E-009-009-MY3, NSC 100-2218-E-009-010-MY3, NSC 100-2219-E-009-011, and NSC 100-2220-E-009-035.

Authors' address: B.-Y. Shen, W.-C. Hsu, and W. Yang, Department of Computer Science, National Chiao Tung University, 1001 University Road, Hsinchu, Taiwan 30010, R.O.C.; email: {byshen, hsu, wuuyang}@cs.nctu.edu.tw.

DOI: http://dx.doi.org/10.1145/2629335

[Smith and Nair 2005], and they have become a standard approach for migrating application binaries from one Instruction Set Architecture (ISA) to another. For example, Apple's Rosetta dynamically translates PowerPC binaries into Intel ×86 binaries, Hewlett-Packard's Aries [Zheng and Thompson 2000] dynamically translates binaries from PA-RISC into IA-64, and DEC developed FX!32 [Chernoff et al. 1998] to make ×86 Win32 applications run on Windows NT/Alpha platforms.

In embedded systems, ARM dominates the market, especially in smartphones and tablets. In order to participate in the fast growing market, many processor vendors such as Intel and MIPS have ported popular embedded operating systems onto their own ISAs. However, they may not have the large number of popular applications ready for their processors as ARM does. Consider that Android has become the worldwide best-selling OS in the smartphone market. And with the release of Jelly Bean (i.e., Android 4.1–4.3), it is further positioned to take control of the smartphone/tablet market. Because of Android's growth, Google has been targeting ARM-based systems for the Android market, which has created some issues with expanding support to other ISAs. One of the main concerns for system vendors is that many popular applications on Android only run on ARM devices. That may seem odd since the majority of these applications are written in Java and are supposed to be independent of specific ISAs. However, for performance or for code reuse reasons, many applications on Google Play contain native libraries in ARM code in their .APK packages. The dependence on such native ARM code makes some of the popular applications unavailable for other ISAs. Without removing dependence on ARM, system vendors with different ISAs are at a disadvantage when competing with ARM-system vendors.

One efficient way to overcome such ISA dependence is to migrate native code to new ISAs using BT techniques [Smith and Nair 2005]. Software-based binary translation systems can be classified in two categories: static (SBT) and dynamic (DBT) [Cifuentes and Malhotra 1996; Altman et al. 2000]. On desktops or workstations, most application binary migrations are based on DBT because DBT can handle the code discovery and the code location problems [Smith and Nair 2005]. However, DBT has its own limitations, such as the overheads of runtime translation and runtime optimizations. Typical DBT systems (e.g., QEMU [Bellard 2005]) use a basic block as a translation unit without performing cross-block optimizations. Some performance-centric DBT systems, such as DynamoRIO [Bruening et al. 2003] and HQEMU [Hong et al. 2012], adopt trace-based code generation with adaptive dynamic optimizations. However, such DBT systems are not suitable for embedded environments since most embedded applications are client-side programs that have relatively short execution time. For short-running applications, these adaptive DBT systems may end up with either full interpretation or fast translated code with minimal optimizations. Furthermore, start-up time and response time are critical to interactive applications, which are common on mobile devices. In addition to the execution time overhead, DBT systems also cause increased memory utilization due to the code cache and the DBT system's own code and data memory. Although this increased memory utilization is small relative to available memory for general-purpose computing environments, it may be extravagant and unacceptable for embedded systems where the memory footprint is a serious concern [Guha et al. 2007; Scott et al. 2001].

On the other hand, SBT must deal with code discovery, code location, and Self-Modifying Code (SMC) issues effectively before being considered as a real solution. The code discovery problem refers to the difficulty of precisely decoding source binaries because data and instructions may be mixed in some sections of the object files. For variable-length ISAs, such as ×86, it is difficult or impossible to even identify the beginning byte of an instruction. For fixed-length ISAs, the code discovery problem is less serious because instruction boundaries are clearly identified. Even if some data

sections are mistranslated into bogus code, they will not get executed at run time. The code location problem is that, for indirect jumps, the jump destination address must be mapped to an address in the translated code. A typical solution for SBT is to implement an Address Mapping Table (AMT) using one entry for each source binary instruction for runtime look-up when indirect branches are executed. This naïve solution may not be acceptable when the size of the application binary is large.

Both the code discovery and code locations problems can be less complex when dealing with fixed-length instructions or variable-length instructions with low variations, such as only 16 bits (short) and 32 bits (long). RISC processors, such as ARM, are based on fixed-length instructions. Although ARM/Thumb interworking can be considered variable-length ISA, it supports only two sizes: 16 bits and 32 bits, so they are easier to handle than completely variable-length ISAs, such as $\times$86. Another well-known limitation of SBT is its inability to handle SMC. However, SMC is not common in embedded applications. We have analyzed over ten thousand applications downloaded from online software stores and failed to find any application that contained SMC. Therefore, we believe that this well-cited problem for SBT should not prevent us from developing useful and practical SBT tools for embedded systems. Furthermore, techniques that statically detect the presence of SMC in a binary [Wang et al. 2008] can be integrated in SBT tools to prevent translating SMC programs.

The LLVM compiler framework [Lattner and Adve 2004] has been used recently in many research and industrial projects. For example, LLVM is used by Android RenderScript to provide platform-independent APIs for high-performance 3D rendering and computing. It is also the core of many OpenCL JIT compilers. In this work, we present a retargetable SBT, LLBT, which leverages the LLVM infrastructure and translates ARM-based binaries into LLVM IR, then retargets to many different ISAs supported by LLVM, such as $\times$86, $\times$86–64, ARM, and MIPS. Because LLVM is already included in Android, it is possible to integrate LLBT into Android devices to perform on-device binary translation. This would allow Android applications with ARM native code to run on devices with ISAs different from ARM.

In summary, this work makes the following contributions:

—We have built a robust *retargetable* SBT that can be used to compare the performance of static and dynamic binary translation approaches for embedded environments. The results show that LLBT-generated code yields more than 6$\times$ higher performance on small and short-running embedded benchmarks and uses less memory than QEMU, which adopts a *retargetable* DBT approach. Compared to a trace-based DBT system, LLBT still yields 2.3$\times$ higher performance than HQEMU on the EEMBC benchmark suite.
—LLBT solves a well-known SBT code location problem [Smith and Nair 2005]. We have come up with innovative AMT optimizations to more effectively handle indirect branch lookups for compiler-generated source binary code.
—In addition to translating fixed-length ISAs, such as ARM and Thumb, LLBT deals with the code discovery problem for ARM/Thumb interworking.
—This article also provides engineering details on how to translate an instruction set into a target-independent compiler IR and how to deal with related issues.

The rest of the article is organized as follows. Section 2 gives an overview of LLBT and provides the details on instruction translation. Section 3 describes our solution to the code discovery problem for ARM/Thumb interworking binaries. Section 4 presents and discusses our results. Section 5 discusses related work, and Section 6 summarizes and concludes.

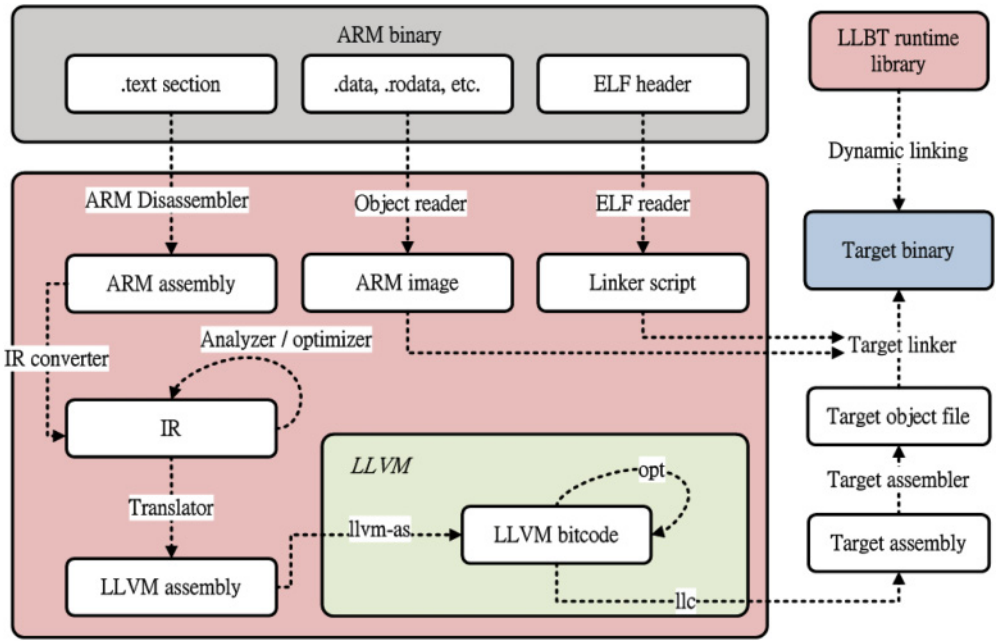

Fig. 1. The architecture of LLBT.

2. LLBT (AN LLVM-BASED STATIC BINARY TRANSLATOR)

LLBT is designed to meet several important requirements for embedded systems. A retargetable BT increases the value of the tool since there are multiple ISAs used in the mobile device market. For example, Android supports devices with ARM, ×86, and MIPS processors. However, crafting a retargetable BT from scratch requires extraordinary effort. In addition to design and implementation of various optimizations, verification would require a lengthy investment. This effort can be significantly reduced if the BT is based on a retargetable compiler framework [Hwang et al. 2010]. LLBT leverages the LLVM compiler infrastructure, which provides a collection of reusable compiler components for supporting both static and dynamic compilation. It translates source instructions into LLVM IRs and then uses LLVM to translate the IRs into target instructions.

For applications running on embedded systems, it is important to start up fast and consume less memory, execution cycles, and energy. To start up fast, we would like to avoid runtime translation overhead. To be space and power efficient, we need more comprehensive and powerful optimization to be performed offline. All these requirements lead us to adopt a static, rather than dynamic, translation approach. In this section, we give an overview of the design and implementation of LLBT and provide the details on how we translate ARM instructions into LLVM IR. In subsequent sections, we discuss how to solve challenging problems for SBT.

2.1. Overview

LLBT takes an ARM binary in ELF format as its input. It translates ARM instructions into LLVM IRs and invokes the LLVM tools to generate optimized code for different target architectures. Figure 1 shows the architecture of LLBT. LLBT is not merely a translator but a tool set that includes a disassembler, an ELF object reader, a translator, a verifier, and runtime libraries. Like a compiler driver, LLBT provides a driver for

```
define i32 @main(i32 %argc, i8** %argv, i8** %envp) nounwind {
entry:
    ; Initialize the emulated ARM registers and stack.
    ......
    ; Set the entry point address of the source binary to the emulated PC register.
    store i32 33024, i32* %ARM_pc, align 4
    ; Branch to the corresponding address of the source entry point.
    br label %L_00008100
    ; Beginning of the source instruction translation.
    ......
L_00008100:
    ......
    ; End of the source instruction translation.
address_translation:
    ; The address mapping table.
    ......
return:
    %ret = load i32* %ARM_r0
    ret i32 %ret
}
```

Fig. 2. An overview of LLVM IR generated by LLBT.

users to perform a number of transformations on the input binary and to invoke target toolchains to generate the target binary. The whole translation process is described as:

(1) An ARM input binary is disassembled to an assembly file, and then an IR converter translates these ARM assembly instructions into LLBT's internal IR. Some analysis and optimization passes, such as identifying PC-relative data and recovering jump tables, will be performed by LLBT on its internal IR before generating the corresponding LLVM instructions.
(2) The LLVM assembly generated by the LLBT translator is assembled into bitcode representation by the LLVM assembler (`llvm-as`). LLVM optimization and analysis passes are selectively performed with the LLVM optimizer (`opt`). The LLVM static compiler (`llc`) performs some target-specific optimizations and transforms the optimized bitcode to a target assembly file.
(3) After generating a target object file, the target linker links the object files with the original ARM's image (excluding text sections) with a linker script. The linker script contains information for the memory layout of the final executable. Each target binary is dynamically linked with an LLBT runtime library, which contains a system call emulator and some helper functions for initialization, profiling, debugging, and dynamic linking-related work.

Figure 2 gives an overview of the LLVM IR generated by the LLBT translator. The `main` function contains all LLVM instructions translated from the source binary. Before the execution jumps to the entry point (i.e., %L_00008100 in Figure 2), the `main` function has to allocate local variables for the emulated ARM architectural state and the source stack. In order to handle command-line arguments and Linux environment variables, LLBT generates calls to helper functions to copy the addresses of arguments and environment variables from the target stack to the source stack. The entry point address of the source binary (i.e., 33024 in Figure 2) is extracted from the ELF header of the source binary.

2.2. Register Mapping

The ARM user mode architectural state includes 16 general-purpose registers and 4 condition flags (i.e., Negative, Zero, Carry, and Overflow). To emulate the execution of ARM binaries, these architecture states are maintained in memory. However, to minimize the generated load/store instructions to access these architecture states,

a BT usually maps such states to the physical registers on the target machine. In the past, many direct DBT systems (e.g., IA-32 Execution Layer [Baraz et al. 2003], IBM's DAISY [Ebcioğlu and Altman 1997], and HP's Aries [Zheng and Thompson 2000]) adopted one-to-one register mapping approaches where one source architecture register is mapped to a unique target register for the entire executable [Smith and Nair 2005], especially when the number of target architecture registers is more than the number of source architecture registers. However, this one-to-one mapping may not work for a retargetable DBT since the number of target registers may be fewer than the source registers. There is no information that source registers may be accessed more frequently to get a higher priority for mapping to limited target registers. In LLBT, we allocate architectural states as LLVM local variables and rely on LLVM's register promotion optimizations to promote local variables to virtual registers and LLVM's global register allocation to map as many virtual registers to physical target registers as possible.

Although LLVM IR provides an infinite number of virtual registers, we cannot map each ARM register to a unique LLVM virtual register directly. This is because the LLVM IR is in SSA form, which means that each update to the architectural state will end up holding the state in a new virtual register. When translating a reference to the architecture state, LLBT must know which virtual registers are possible definitions, and `phi` nodes must be properly inserted. This complicates the translation and makes the translated code difficult to understand. Instead, LLBT declares ARM registers and condition flags as local variables, which are allocated by the LLVM `alloca` instruction. The LLVM optimizer will take care of promoting local variables from memory references to register references and then map them to physical registers through the global register allocation phase. In LLBT, many of the ARM registers and flags end up mapped to the physical registers of the target machine. Comparing LLBT to a retargetable DBT, such as the QEMU, we have observed a huge difference in the number of load/store instructions generated in accessing the source architecture states. The just described register mapping strategy used in LLBT contributes significantly to the performance advantage of SBT.

2.3. Instruction Translation

In LLBT, instruction translation is a one-to-many mapping process between ARM and LLVM instructions. The process can be divided into three phases:

(1) **Conditional execution check** examines the condition flags in the current program status register (`CPSR`) to determine whether the instruction should be executed. In the ARM ISA, almost all instructions can be conditionally executed. As shown in Figure 3(3), LLBT translates a conditional execution instruction into three parts. The first part checks the condition flags. If the condition check succeeds, the control will flow to its instruction body. Otherwise, the control will flow to the next instruction. Instructions that are not conditionally executed will be omitted, and the translated LLVM IRs will begin at their respective instruction bodies.
(2) **Instruction body** implements main operations of ARM instructions, including updating the ARM architectural state and transferring control flow.
(3) **Condition flag update** is important for comparison instructions, such as `cmn` (Compare Negative), `cmp` (Compare), `teq` (Test Equivalence), and `tst` (Test), and more. In addition, some other arithmetic, logical, and move instructions with the `S` qualifier [Seal 2000] will also update the condition flags. For the aforementioned ARM instructions, LLBT generates instructions for updating the condition flags.

Figure 3(1) and Figure 3(3) show an ARM `ands` instruction and the corresponding LLVM IRs, respectively. The detailed operations of the instruction are listed in

(1): ARM Instructions:

```
andsne r1, r1, r2
```

(2): Operations of ANDS:

```
Instruction
    ANDS Rd, Rd, Rm

Operation
    Rd = Rd AND Rm
    N Flag = Rd[31]
    C Flag = unaffected
    Z Flag = if Rd == 0
             then 1 else 0
    V Flag = unaffected
```

(3): LLVM Instructions:

```
Conditional execution check.
 1. L_1:
 2.   %0 = load i32* %CPSR_z
 3.   %1 = icmp eq i32 %0, 0
 4.   br i1 %1, label %L_2, label %L_3
Instruction body.
 5. L_2:
 6.   %2 = load i32* %ARM_r1
 7.   %3 = load i32* %ARM_r2
 8.   %4 = and i32 %2, %3
 9.   store i32 %4, i32* %ARM_r1
Conditional flags update.
10.   ; Update the N flag.
11.   %5 = load i32* %ARM_r1
12.   %6 = lshr i32 %5, 31
13.   store i32 %6, i32* %CPSR_n
14.   ; Update the Z flag.
15.   %7 = load i32* %ARM_r1
16.   %8 = icmp eq i32 %7, 0
17.   %9 = select i1 %8, i32 1, i32 0
18.   store i32 %9, i32* %CPSR_z
19.   br label %L_3
The next instruction.
20. L_3:
```

Fig. 3. An instruction translation example. (1) An ARM ands (Logical AND) instruction with the NE (Not Equal) condition. (2) The detailed operations of the ands instruction. (3) The corresponding LLVM instructions.

Figure 3(2). In Figure 3(3), lines 1–4 check whether the Z flag in CPSR register satisfies the specified condition (i.e., NE). If satisfied, the instruction body (i.e., lines 5–9) will be executed, and the condition flags will be updated (i.e., lines 10–19). Otherwise, a direct branch that jumps to the next instruction (i.e., line 20) will be taken.

2.4. Handling Indirect Branches

Unlike direct branches, the destination address of an indirect branch is not known until runtime. In order to handle indirect branches at translation time, LLBT prepares an Address Mapping Table (AMT) that maps each source address of an indirect branch destination to the corresponding destination address in the translated code and generates a jump for the indirect branch instruction. The jump instruction transfers control to a stub that searches the AMT for the destination address. A naïve AMT will be very large because it must include an entry for every ARM instruction assuming each instruction could be a jump target.

In order to reduce the size of the address mapping table, LLBT does not keep an entry for every source instruction. Instead, it maintains entries for the source instructions that can possibly be destinations of an indirect branch for compiler-generated instead of hand-crafted binaries, which include (1) return addresses, (2) function entry points, and (3) the addresses stored in jump tables. In this subsection, we discuss how to find return addresses and function entry points. Handling jump tables will be discussed in Section 3.2.

2.4.1. Return Address. Finding return addresses is relatively easy because a return address is the address of the instruction that immediately follows a function call. We identify function call instructions in the binaries to find the return addresses.

2.4.2. Function Entry Points. Function entry points can be found in the symbol table of the input binary. However, if the symbol table is stripped off, LLBT uses the following steps to locate possible function entry points:

(1) ARM Instructions:

```
mov pc, r3
```

(2) LLVM Instructions:

```
%0 = load i32* %ARM_r3
store i32 %0, i32* %ARM_pc
br label %address_translation
```

(3) Address Mapping Table:

```
address_translation:
  %address = load i32* %ARM_pc
  switch i32 %address, label %not_found [
    i32 32120, label %L_00007d78
    i32 32160, label %L_00007da0
    ....
    i32 46360, label %L_0000b518
  ]
not_found:
  ......
```

Fig. 4. (1) An example of indirect branch instructions and (2) the corresponding LLVM instructions. (3) An example of the address mapping table.

—**Inspecting direct function call instructions**. Although function calls through a direct branch instruction, such as `bl` in ARM ISA, do not need to search the address mapping table at runtime, the function address may also be a possible destination address of other indirect function call instructions, such as `blx` in ARM ISA.
—**Examining all the PC-relative load instructions in input binaries**. If the loaded data are an address located in the code segment, it is likely to be a function entry address used by an indirect function call instruction.
—**Examining the data section of input binaries**. A common example of storing function entry points in the data section is the virtual method table, which is a mechanism used to support dynamic dispatch for some programming languages like C++. Programmers may also store function entry addresses in a global function pointer branch table for implementing dynamic dispatch.
—**Collecting instruction addresses that immediately follow a return instruction**. PC-relative data that immediately follow a `return` instruction will be skipped, and the next instruction is considered instead.

2.4.3. Address Mapping Table (AMT). Figure 4 show an example of indirect branches and the AMT. LLBT will update the value of `ARM_pc` and jump to `address_translation` if the instruction is an indirect branch. The AMT is implemented by the LLVM `switch` instruction. If the value of `ARM_pc` can be found in the switch cases, the program will jump to the corresponding destination. Otherwise, the control flow will be transferred to the default destination, `not_found`, and the control will be handed to the LLBT runtime manager.

The LLVM `switch` instruction may be implemented in different ways, depending on target machines and the instruction context. Since the AMT consists of sparse case values, LLVM is likely to translate the `switch` instruction into a series of conditional branches. Such an implementation may substantially decrease the performance of table lookup, especially when the table is large. Therefore, to speed up the table search, we currently use a runtime dispatcher to hash the source address (i.e., `ARM_pc`) to one of many smaller AMTs.

2.4.4. Hand-crafted Assembly Code. LLBT is designed for compiler-generated code, and thus it may have problems dealing with tricky indirect branch targets possibly set by assembly programmers. However, as mentioned earlier, for applications downloaded from online stores, there are two primary reasons that native code may be embedded in the .APK packages: (1) for code reuse and (2) for high performance.

(1) Code reuse is usually the primary reason to use native code because software vendors can integrate applications with their existing low-level (C/C++) APIs, which have been fine-tuned and thoroughly tested, without developing a new one [Lee and Jeon 2010; Kurzyniec and Sunderam 2001]. For the Android applications to

ARM Instructions:

```
bl __aeabi_idiv
```

LLVM Instructions:

```
%0 = load i32* %ARM_r0
%1 = load i32* %ARM_r1
%2 = sdiv i32 %0, %1
%3 = srem i32 %0, %1
store i32 %2, i32* %ARM_r0
store i32 %3, i32* %ARM_r1
```

Fig. 5. Integer division helper replacement.

use those existing library functions, developers may include ARM binaries for the Dalvik Virtual Machine (DVM) to call such functions via JNI interface. Such native code is generated by compilers, not manually coded.

(2) Although native code might be included for performance reasons, it does not necessarily need to be hand-coded assembly instructions. For example, the currently released JIT in DVM is trace-based, compiled native code (Ahead-of-Time) that could avoid overheads of DVM interpretation and dynamic compilation from the trace-based JIT and benefit from more aggressive method-based optimizations [Wang et al. 2011]. Another example is that developers may write C/C++ code to manage memory allocation/deallocation for their applications rather than relying on the JVM garbage collector [Lee and Jeon 2010].

When LLBT disassembled the native code enclosed in 1,000 downloaded .APK packages, most of the native codes are generated by compilers. Although it is hard to tell whether in-line assembly was used in the original source code, we believe it is very unlikely that programmers would use in-line assembly to set arbitrary branch targets for indirect branches. Even if native code does contain hand-crafted assembly code with an indirect branch destination, a component in our runtime system [Shen et al. 2012] serves as a safety net to handle such exceptional cases.

2.5. Helper Function Replacement

Because many ARMv5 implementations do not have an FPU, in order to maintain compatibility, applications usually use a floating-point library to emulate floating-point operations, especially for online store applications such as Android applications. For a similar reason, because ARMv5 does not provide an integer division instruction, compilers generate a sequence of code that invokes an external helper function for integer division.

ARM defines a set of runtime helper functions, which are implemented in a compiler's runtime library (such as `libgcc`) for emulating floating-point and integer division operations. The floating-point helper functions use software floating-point calling conventions even when hardware implemented floating-point instructions are available on the target device [ARM Limited 2012]. Because these helpers are target-specific and other CPUs may have integer division or floating-point instructions, if we migrate an ARM binary to other targets, we have to either (1) translate each ARM instruction in the helpers or (2) implement these helpers using target instructions. LLBT adopts the second approach for performance. In LLBT, call instructions that invoke an external helper will be replaced by a sequence of LLVM instructions. Figure 5 shows an ARM instruction that invokes the integer division helper and the corresponding LLVM instructions that replace the helper call. After the helper function replacement, LLVM could generate integer divide and/or FP instructions directly on the target machine.

2.6. Debugging Support and Verification

LLBT utilizes the `metadata` feature in LLVM to include source binary information in the target binary. Such `metadata` can be used to generate debugging information in

target binaries so that the generated target binary can be debugged with debuggers such as `gdb` and `lldb`. For example, we can use the `step` command in `gdb` to single-step through the execution of the source instructions in the target binary. We can also print the values of source registers and condition flags when using `gdb` to debug the target binary generated by LLBT. Furthermore, LLBT innovates on automatic verification of the translated binary code, making the development of SBT tools less of a nightmare. We designed and implemented a mechanism that can automatically compare the execution steps of the source and translated programs. However, details on how to automatic verify LLBT-generated code are outside the scope of this article and can be found elsewhere [Chen et al. 2013a].

3. CODE DISCOVERY FOR ARM/THUMB INTERWORKING BINARIES

The code discovery problem refers to the difficulty of precisely decoding binaries because data and instructions may be mixed in the text. For ISAs with a fixed instruction size, this is less of an issue because all instruction boundaries can be clearly identified. Even if data were misinterpreted as instructions, they will not get executed at runtime. The original ARM and Thumb architectures are both fixed-size ISAs where ARM instructions are 32-bit wide and Thumb instructions are 16-bit wide. However, since ARM7TDMI, ARM has started supporting interworking between ARM and Thumb states, which means ARM and Thumb instructions can coexist in the same binary. When in the ARM state, the processor executes ARM instructions, and in the Thumb state, the processors executes the Thumb instructions. State switching between ARM and Thumb is based on the execution of `bx` and `blx` branch instructions. The least significant bit of the branch address will determine whether the target instruction is ARM or Thumb. For direct branches, the target address is known at translation time, and the state of the branch target can be determined. However, for the indirect branches, the branch target address can only be known at runtime, and this would prevent the SBT from identifying states of all regions. Therefore, in ARM binaries mixed with Thumb instructions, the disassembly results may be incorrect unless the translator knows in which state the instruction would be executed.

Normally, the ARM toolchain generates some special symbols in ARM binaries that can help identify ARM, Thumb, and data regions in the binaries. These symbols include `$a`, `$t`, and `$d`, each representing the beginnings of ARM instruction regions, Thumb instruction regions, and data regions, respectively. With these symbols, every word in the identified region can be disassembled correctly. However, stripped binaries do not have symbol tables, which means the programs lack special symbols and cannot be separated into different regions. The following subsections describe how we solve the code discovery problem in SBT for ARM/Thumb interworking binaries.

3.1. ARM/Thumb Region Identification

For stripped binaries where the `$a`/`$t`/`$d` symbols are not available, LLBT must identify the ARM/Thumb/Data regions. LLBT requires a set of entries as the starting points of code discovery. In a stripped executable, the program entry point is still available. In a stripped shared object, the dynamic symbol table contains all exported function symbols used for dynamic linking. The ARM/Thumb ISA type can be determined by the last bit of the entry address. Zero means the entry is an ARM instruction, and one denotes a Thumb instruction. With these entries, LLBT can discover ARM/Thumb regions in two steps: (1) enclose a region for each entry, (2) discover more entries through enclosed regions. The two steps are elaborated as:

(1) **To enclose a region for an entry:** LLBT will check each instruction that can be reached from the entry through a sequence of straight-line code until an

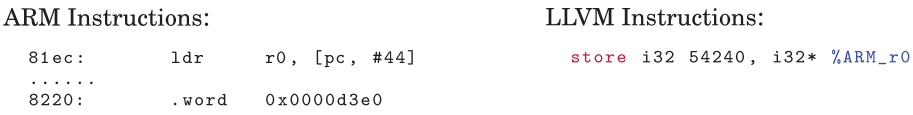

Fig. 6. An example of inlining PC-relative data.

unconditional branch is found. This branch is considered the end of the region enclosed by the entry. Since all the instructions in the region may be executed starting from the entry without changing states, the instructions in the same region must belong to the same ISA.

(2) **To discover more entries:** Once a region is discovered, the direct branches, such as `b` and `bl` instructions, encountered in the region must jump to regions of the same ISA. The direct branches that can change instruction sets, such as `blx`, can help to find entries different from the current ISA. The two steps will be repeated until no new entries are found.

For each of the remaining unknown regions, LLBT will translate it as both an ARM region and a Thumb region. During runtime, one of them will be selected based on the address of the branch register. For the discovered region, translating a single ISA improves the time and space efficiency. We also developed some methods to analyze the remaining unknown regions for identifying more ARM/Thumb regions [Chen et al. 2013b]. In our experiments, this code discovery method can identify more than 95% of ARM/Thumb regions in binaries with ARM/Thumb interworking.

In addition to Thumb ISA, a new ISA of ARM, Thumb-2, was introduced in the ARM1156 (ARMv6) core. Thumb-2, a superset of the Thumb ISA, has performance close to ARM ISA and code density similar to Thumb ISA. Unlike Thumb ISA, which has only 16-bit instructions, Thumb-2 allows 32-bit instructions to be mixed with 16-bit Thumb instructions without switching states. The current approach of identifying ARM/Thumb regions can also be used to identify ARM/Thumb-2 regions. In Thumb-2, 32-bit instructions can be distinguished from 16-bit instructions based on instruction encoding. However, the main challenge is how to separate data regions from instruction regions. If data were misinterpreted as instructions in Thumb-2, the instruction decoding in the following text may be wrong. The subsequent subsection discusses the general approach used in LLBT.

3.2. Discrimination between Data and Instructions

Because LLBT is designed for translating compiler-generated code rather than hand-crafted binaries, only limited kinds of data may be embedded in the text section. In the mainstream compilers, such as GCC and LLVM/Clang, there are two kinds of data in the text section: (1) PC-relative data and (2) jump tables.

3.2.1. PC-relative Data. Due to ARM's small 8-bit immediate fields, compilers typically place constants in a literal pool and generate a PC-relative load instruction to move the constant from the literal pool into a register when the constant is too large to fit in the immediate field. Once we can identify ARM/Thumb regions in the binaries, finding PC-relative data is easy because we recognize all PC-relative load instructions. A constant that cannot fit in the immediate field in ARM ISA might be encoded as immediate values in the target ISA, which supports larger immediate fields. For example, ×86 supports 32-bit immediate operands. Therefore, LLBT will always copy the corresponding constants of all PC-relative load instructions and inline them into LLVM `store` instructions. This transformation, called PC-relative data inlining, provides an opportunity for the LLVM to encode more constants in target instructions. Figure 6 shows an example of inlining PC-relative data.

(1) ARM Instructions:

```
83ec:       cmp     r5, #4
83f0:       ldrls   pc, [pc, r5, lsl #2]
83f4:       b       8454
83f8:       .word   0x0000840c
83fc:       .word   0x00008414
8400:       .word   0x00008424
8404:       .word   0x00008434
8408:       .word   0x00008444
......
```

(2) LLVM Instructions:

```
L_000083f0:
%0 = load i32* %ARM_r5
%1 = shl i32 %0, 2
%2 = add i32 %1, 33784 ; index + ARM_pc
switch i32 %2, label %L_00008454 [
  i32 33784, label %L_0000840c
  i32 33788, label %L_00008414
  i32 33792, label %L_00008424
  i32 33796, label %L_00008434
  i32 33800, label %L_00008444
]
```

Fig. 7. (1) An example of a jump table in ARM instructions generated by GCC and (2) the corresponding LLVM `switch` instructions recovered by LLBT.

3.2.2. Jump Tables. Compilers commonly use jump tables to implement `switch` statements. For example, Figure 7(1) shows an example of jump tables generated by GCC. An indirect branch is used to access the branch targets from a jump table. As shown in Figure 7(1), the destination address of each case in the `switch` statement is stored as PC-relative data. The AMT generated by an SBT include each address in the jump table because it is a branch destination of an indirect branch. However, it is possible to revert a jump table to a switch statement.

Cifuentes [Cifuentes and Emmerik 1999] proposed a technique to recover jump tables in a machine-independent way that can find more than 90% of addresses in jump tables. Another common way to overcome this problem is to use patterns generated from a particular set of compilers [De Sutter et al. 2007; Chanet et al. 2007]. In LLBT, we employ the second approach since most applications downloaded from online stores are developed by using official toolchains, such as Android NDK. Figure 7(2) shows the LLVM `switch` instructions recovered from Figure 7(1) by LLBT. Once the jump tables are recovered, the destination addresses of indirect branches in the AMT will be limited to function entry points and return addresses. This would also reduce the size of AMTs. In addition, this transformation can also speed up the AMT lookup and enable LLVM to generate a jump table in target binaries.

3.2.3. Source Text Section Removal. With the help of PC-relative data inlining and jump table recovery, we no longer need the text section of input binaries in the target binaries. This is because all instructions in the text section have been translated, and all data in the text section have been inlined/recovered to LLVM instructions. This optimization can reduce the code size for embedded systems. In comparison, DBTs have to keep the text section in memory because source instructions are translated at runtime.

4. EXPERIMENTAL RESULTS

We have evaluated LLBT across more than 40 industry standard benchmarks, including SPEC CPU2006 and EEMBC 1.1. Our experiments include performance compared to native compilation and DBT, LLVM optimization analysis, start-up time, space overhead from AMTs, code size measurement, runtime memory overhead, and translation time. The ARM binaries (i.e., source binaries) of the benchmarks were compiled with the GNU GCC 4.7 compiler. The cross-compiler is configured to generate ARMv5TE instructions with software floating-point operations since this is the default configuration in Android NDK. The LLVM version used in our experiments is 3.2. The target binaries translated by LLBT use the LLVM optimization setting in level 2 (i.e., *-O2*).

4.1. Binary Translation vs. Native Compilation

To show performance relative to native compilation on the target, we first compared the source binaries and target binaries on the same architecture. We used LLBT to

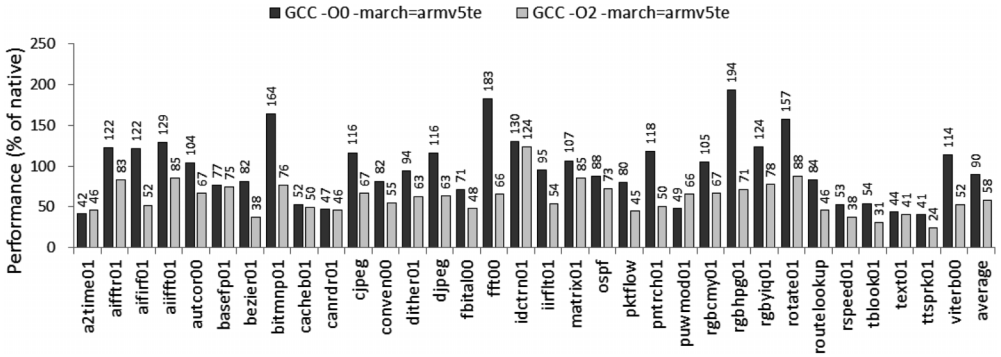

Fig. 8. Performance of LLBT on translating the EEMBC benchmark suite on an ARM926EJ-S processor. The performance is relative to its source binary running on the same processor. These benchmarks were compiled by GCC with the optimization options "*-O0*" and "*-O2*."

translate ARM binaries into LLVM IR and then used LLVM to compile the IR back to the same ARM architecture (i.e., ARMv5TE). This experiment can uncover possible significant inefficiencies introduced by the translation.

Figure 8 shows the experimental results of the EEMBC benchmark suite on an ARM926EJ-S processor. On average (geometric mean), the translated binaries obtain 90% of the original performance if the original binaries are unoptimized (*-O0*), and 58% of the original performance if the original binaries are optimized (*-O2*). For unoptimized binaries, LLBT outperforms original source binaries because LLBT can perform global optimizations on LLVM IR. Surprising, translated binaries could occasionally outperform natively compiled and optimized binaries. For example, on the `idctrn01` benchmark, the translated binary reached 124% performance of the original binary.

Therefore, there may be still some optimization opportunities in the optimized binary. Moreover, some optimizations, such as loop unrolling and loop unswitching, enabled in LLVM's "*-O2*" are not enabled in GCC's "*-O2*." If we compiled `idctrn01` with GCC's optimization options "*-O2 -funroll-loops*," the performance of the translated code would be reduced to 81% of the original binary. Another reason is that LLBT translates a single fully linked binary rather than separately compiling source files into object files and linking the object files into a single executable. This provides additional optimization opportunities that are only available at link time in native compilation. If we enabled link-time optimization for GCC, the performance of natively compiled EEMBC would improve by about 10%, on average.

Figure 9 shows another set of experimental results on an Intel Atom D2700 processor. We compared performance of the binaries generated by LLBT with binaries that are directly compiled from C source programs by LLVM's clang compiler. The optimization setting on LLBT and LLVM's clang compiler are the same (*-O2*), which means the LLVM IRs generated by LLBT and LLVM's clang compiler apply the same optimizations and use the same compiler backend. We used LLBT to translate the original ARM binaries into ×86 and ×86–64 binaries, respectively. The performances are 55% and 60% on average on ×86 and ×86–64, respectively. From Figure 9, we can see that, on ×86–64, LLBT can outperform native compilers on some benchmarks, such as `bezier01`, `idctrn01`, and `iirflt01`. The higher performance on ×86–64 is because the `long` data type in GCC and LLVM's clang is 4 bytes on ARM and ×86 but 8 bytes on ×86–64. Therefore, the memory footprint of natively compiled ×86–64 binaries is much larger than that of the corresponding ×86–64 binaries translated from ARM.

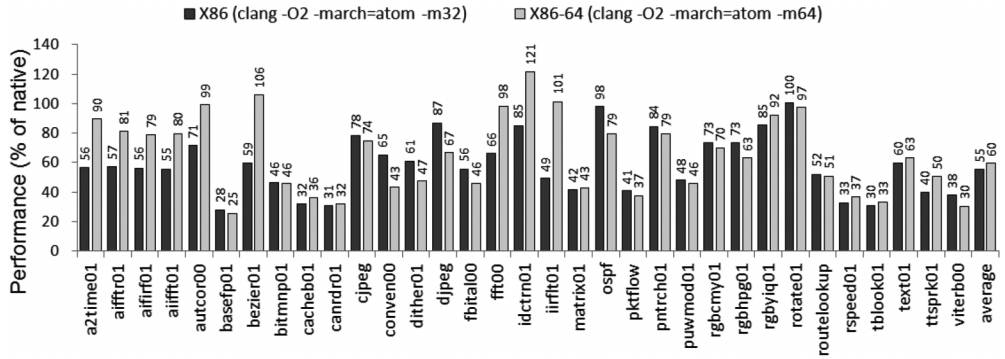

Fig. 9. Performance of LLBT on translating the EEMBC benchmark suite from ARM to ×86 and ×86–64 on an Intel Atom D2700 processor. The performance is relative to natively compiled binaries that were compiled by LLVM's clang compiler from C source code into ×86 with "*-O2 -march=atom -m32*" options and into ×86–64 with "*-O2 -march=atom -m64*" options.

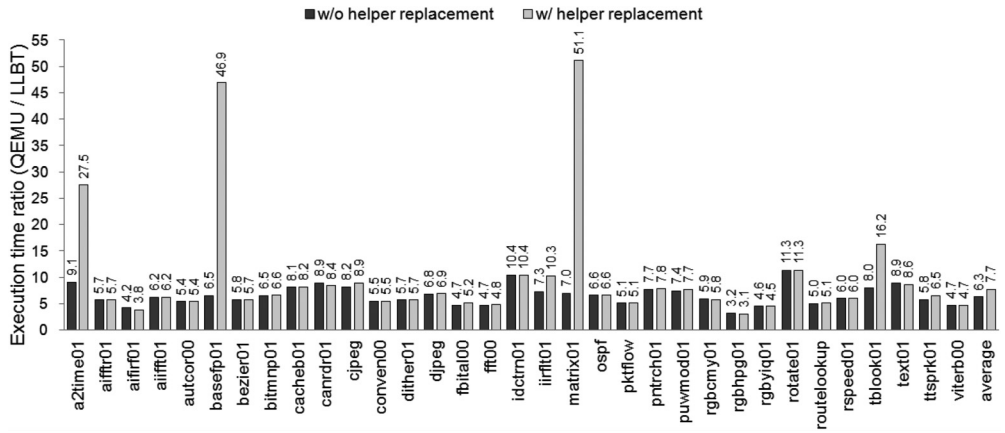

Fig. 10. Execution time ratio of "QEMU / LLBT" on translating the EEMBC benchmark suite from ARM to MIPS on an Ingenic JZ4760 MIPS processor. These benchmarks were translated by LLBT with and without the helper function replacement.

4.2. SBT vs. DBT

Next, we would like to compare the performance difference between static and dynamic binary translations. In this subsection, we first compare LLBT with QEMU running the EEMBC benchmark suite on a MIPS machine. Second, since the EEMBC benchmark suite represents small and relatively short-running applications, an evaluation of translating SPEC CPU2006 benchmarks on ×86 is included because they are long-running and CPU intensive. Finally, we compared LLBT with HQEMU [Hong et al. 2012], which is a trace-based and highly optimized DBT system, on a ×86–64 machine. All benchmarks were compiled by GCC with the *-O2* optimization setting.

4.2.1. LLBT vs. QEMU on Small and Short-Running Embedded Benchmarks. Figure 10 shows the results of the EEMBC benchmark suite running on an Ingenic JZ4760 MIPS processor. We used LLBT to translate EEMBC from ARM to MIPS with and without enabling the helper function replacement. We then compared the performance of

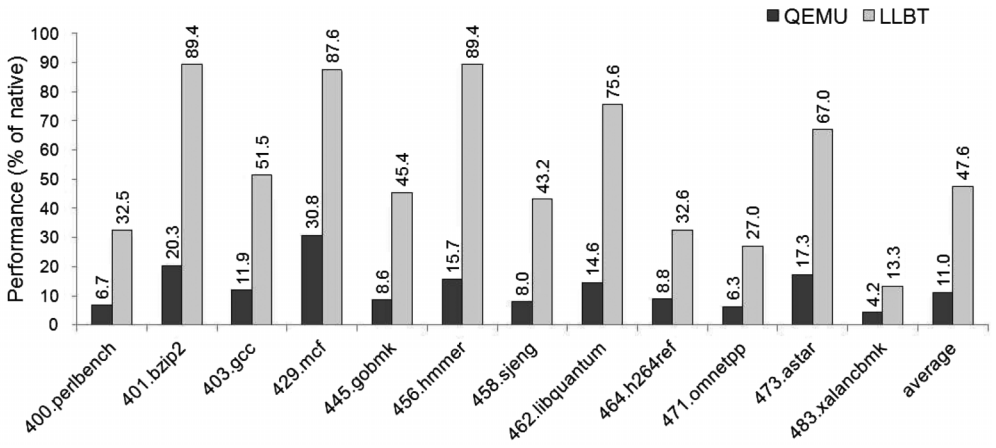

Fig. 11. Performance of LLBT and QEMU on translating SPEC CPU2006 integer benchmarks from ARM to ×86 on an Intel Xeon E5506 processor with reference inputs. The performance is relative to natively compiled binaries that were compiled by GCC with the *-O2* option.

translated binaries with that of ARM binaries on the QEMU user mode emulator on the same platform. The QEMU used in our experiments is version 1.2.2. The performance comparison shows that LLBT-generated code can run 6.3× faster than that generated by QEMU and 7.7× faster when the helper function replacement is enabled. On some benchmarks, such as `a2time01`, `basefp01`, `iirflt01`, `matrix01`, and `tblook01`, the outstanding performance improvement from the helper function replacement is because they are floating-point intensive and spend significant time in floating-point helpers. When we took out the five floating-point intensive benchmarks, the geometric means are reduced from 7.7× to 6.2×. The result is similar to disabling floating-point function replacement. Our static translated code still runs much faster than running on QEMU, which adopts retargetable DBT approaches.

4.2.2. LLBT vs. QEMU on Long-Running and CPU-Intensive Benchmarks. Figure 11 shows the experimental results of SPEC CPU2006 running on an Intel Xeon E5506 processor. The performance comparison shows that LLBT-generated code obtains 47.6% performance of native execution on average. In addition, LLBT-generated code can still run 4.3× faster than QEMU on these large and long-running benchmarks. In Figure 11, some benchmarks, such as `429.mcf` and `483.xalancbmk`, do not have much performance improvement compared to QEMU. The lower performance improvement on `429.mcf` is because QEMU runs relatively fast on this benchmark. Besides, compared to natively compiled code, the performance of LLBT-generated code is close to 90% of native execution on `429.mcf`. Therefore, there may not be much potential performance improvement on `429.mcf` for LLBT. For `483.xalancbmk`, the lower performance improvement is due to a large number of indirect branches and indirect branch destinations. In addition to an AMT lookup, the translation of an indirect branch also involves some register remapping. This is because an indirect branch may jump to any address in the AMT, and the same emulated ARM registers may be allocated to different physical registers or spilled to memory according to their usage in different basic blocks. While the table lookup is already implemented as a kind of a hash table, the register remapping may involve multiple memory load/store. Therefore, an indirect branch is memory intensive. This is happening not only in SBT, but also DBT, because indirect branches in DBT result in switching back to the runtime emulator and have to use register remapping for the context switch.

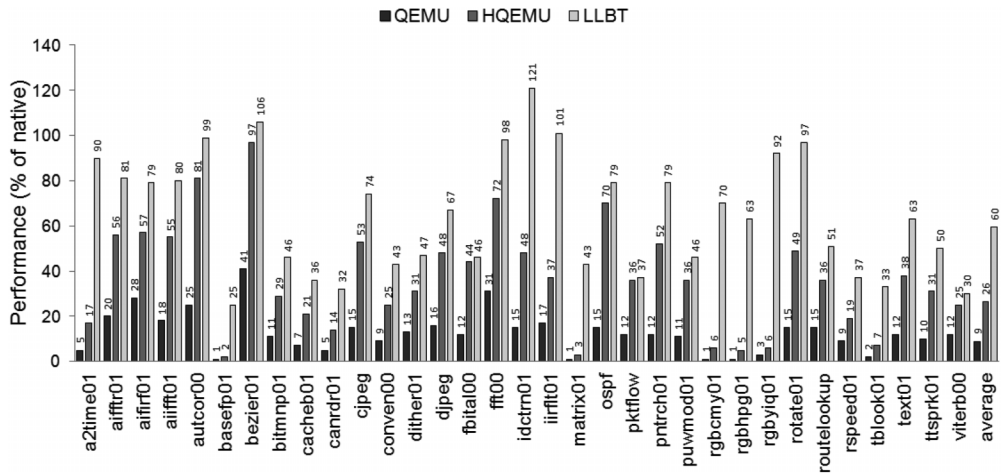

Fig. 12. Performance of QEMU, HQEMU, and LLBT on translating the EEMBC benchmark suite from ARM to ×86–64 on an Intel Atom D2700 processor. The performance is relative to natively compiled binaries which were compiled by LLVM's clang compiler from C source code into ×86–64 with '*-O2 -march=atom -m64*' options.

4.2.3. LLBT vs. HQEMU. HQEMU is a trace-based DBT system that runs QEMU's translator and an LLVM translator with a dynamic binary optimizer on different threads. In addition to merging small translation blocks into traces, HQEMU can also dynamically combine traces into larger ones to enlarge the optimization scope and reduce redundant memory operations during trace transitions [Hong et al. 2012]. As a result, HQEMU can run 2.4× faster than QEMU for the SPEC CPU2006 integer benchmarks on translating ARM into ×86–64, while our previous experiment shows that LLBT-generated code can run 4.3× faster than QEMU. In this subsection, we compared LLBT with HQEMU on the EEMBC benchmark suite.

Figure 12 shows the experimental results running on an Intel Atom D2700 processor. Since HQEMU can only perform ARM to ×86–64 and ×86 to ×86–64 emulation currently, we compared LLBT with HQEMU on translating ARM into ×86–64. The result of QEMU running on the same platform is also included in this figure for comparison. As shown in Figure 12, compared to QEMU, the trace-based code generation and dynamic optimizations of HQEMU do improve the quality of translated code significantly. However, the average performance of LLBT-generated code is still 2.3× faster than HQEMU.

In summary, SBT translated binaries run faster than DBT translated binaries for two main reasons: (1) DBT has runtime overhead, including interpretation, translation, and runtime management. This overhead is more serious for programs with relatively short runtimes. (2) DBT performs much less code optimization since optimization time is part of runtime. For long-running applications, for which the translation overhead can be amortized, the main difference in performance would be due to the quality of translated code. QEMU does not perform cross-block optimizations, whereas LLVM is rather strong in global optimizations. This gap can be reduced in the future because new DBT have adopted trace-based code generation [Hong et al. 2012] and optimization rather than the one-block-at-a-time approach used by the current QEMU.

4.3. LLVM Optimization Analysis

One of the important functionalities that LLBT leverages from LLVM is comprehensive optimizations. In order to know which optimizations contributed most to the

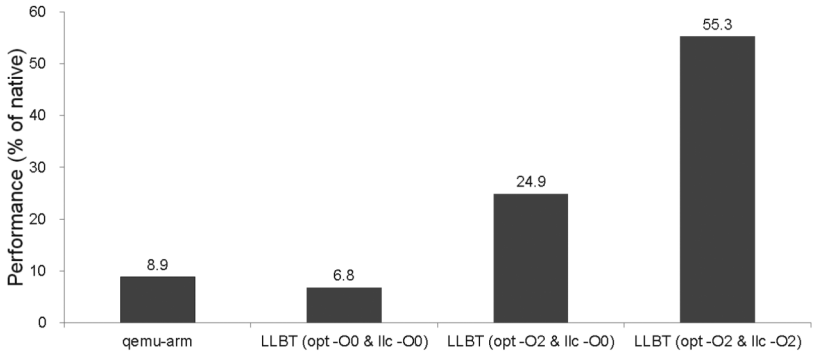

Fig. 13. Comparing the average performance of the EEMBC benchmark suite translated by LLBT in different LLVM optimization levels. `opt` is the LLVM optimizer, which is used to perform target-independent analyses and optimizations. `llc` is the LLVM static compiler.

performance improvement and why such optimizations were not applied in DBT, we analyzed the optimizations invoked in LLVM. In LLVM, the optimizer (i.e., `opt`) and the static compiler (i.e., `llc`) can perform target-independent and target-dependent optimizations on LLVM IR, respectively. Therefore, we analyzed them separately.

Figure 13 shows the experimental results of the EEMBC benchmark suite running on an Intel Atom D2700 processor. The $\times$86 binaries were translated under different optimization levels to show the effects of LLVM optimizations. In the first configuration, all LLVM optimizations are disabled. The average performance of the translated code is a little slower than running on QEMU. In this configuration, the LLVM static compiler used local register allocation and a fast instruction selection algorithm to speed up code generation. This behavior is somewhat similar to other retargetable DBT systems, such as Strata [Scott et al. 2003], which use a basic block as a translation unit and have to load/store the architectural state at the translation block boundaries.

In the second configuration, we turn on the LLVM target-independent optimization (*-O2*) to enable several time-consuming optimizations. The result shows that LLBT translated code can be 2.8$\times$ faster than the QEMU runs. In the final configuration, we turn on target-dependent optimizations and also enable global register allocation on the whole translated IR. The performance of our SBT system can now be more than 6$\times$ faster than the QEMU run.

As a result, LLVM contributes about 3.7$\times$ and 2.2$\times$ performance improvement to the target binaries on target-independent optimizations and target-dependent optimizations, respectively. After further investigation on the LLVM static compiler, we found that the highest performance improvement on target-dependent optimizations comes from global register allocation, which contributes about 2$\times$ performance compared with local register allocation. On the other hand, there are many target-independent optimizations in LLVM. In order to know which optimizations contributed most to the performance improvement, we analyzed the optimizations executed by the LLVM optimizer in the optimization level 2 (i.e., *-O2*).

Because one optimization may provide additional opportunities for other optimizations, some optimizations execute more than once. For example, in the LLVM optimizer, the *simplify the control flow graph (CFG)*, which performs dead code elimination and basic block merging, may execute more than three times, and the *combine redundant instructions* may execute more than four times. Running optimizations in different orders may also affect the final results. Therefore, we use a subtraction mechanism that removes optimization passes in LLVM's optimization level 2 one by one to observe the performance impact.

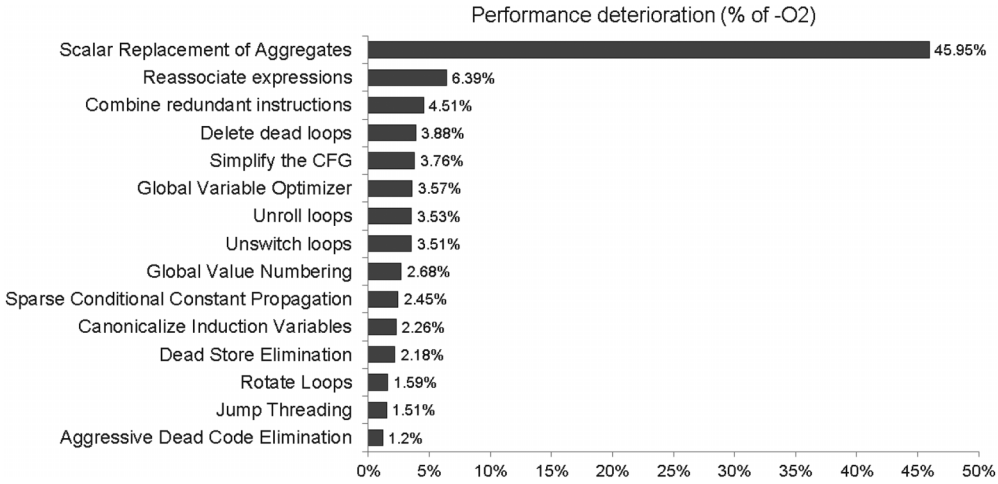

Fig. 14. Comparing the average performance deterioration of removing each optimization executed by the LLVM optimizer in optimization level 2 (i.e., *-O2*) on the EEMBC benchmark suite. Optimizations without noticeable performance difference are removed in this figure.

Figure 14 shows the average performance deterioration of removing each of LLVM's optimization on the EEMBC benchmark suite. From the figure, we can see that most optimizations provide less than 7% improvement except the *scalar replacement of aggregates*, which transforms each member in an aggregate type (e.g., a structure or an array) into individual variables, if possible. This optimization can make members in an aggregate type become the candidates for register allocation, constant propagation, copy propagation, and other optimizations that apply to scalars [Muchnick 1998]. Because no type information of source languages is retained in the source binaries, the LLVM IR generated by LLBT has no type information either. The scalar replacement comes from the types for emulating the ARM architectural state. The types are as follows:

```
; 16 general-purpose registers
%struct.regType = type { i32, i32, i32, i32, i32, i32, i32, i32,
                         i32, i32, i32, i32, i32, i32, i32, i32 }
; 4 conditional flags in CPSR
%struct.psrType = type { i32, i32, i32, i32 }
```

Therefore, the performance improvement from LLVM's target-independent optimizations depends on how LLBT generates LLVM IR. The performance deterioration caused by removing the *scalar replacement of aggregates* occurs because it prevents mapping the emulated ARM architectural state to the physical registers on the target machine. Furthermore, since LLBT translates source instructions into LLVM IRs in a single LLVM `main` function, LLVM's global register allocation can map emulated ARM registers to physical target registers according to how often the emulated ARM registers are accessed in whole source binaries. As a result, the highest performance improvement contributed by LLVM comes from mapping as many emulated ARM registers to physical target registers as possible and minimizing the generated load/store instructions to access emulated ARM registers if there are not enough physical target registers.

4.4. Start-Up Time

Fast start-up is important for embedded applications. In this subsection, we compared the start-up time of a statically translated program with dynamically translated ones.

We used a real-world program, Lynx (version 2.8.7), as our benchmark. Lynx is a text-based web browser that is one of the oldest web browsers still in general use and development. In order to eliminate network latency, we measured the execution time from beginning until opening a blank page at localhost. The dynamic translators we use in the comparison are QEMU and HQEMU, which are an unoptimized DBT system and an optimized trace-based DBT system, respectively. The experiment was run on an Intel Atom D2700 processor.

The result shows that Lynx emulated by QEMU requires about 10.3× and 8.1× the start-up time of the LLBT translated ones on ×86 and ×86–64, respectively. The gap between the two is caused by the large amount of translation overhead at the beginning of a process. After profiling on QEMU, we found that only about 20% of the start-up time is spent in the code cache. The high translation overhead at the beginning of the process implies that there might be no hot spots in the execution paths at the start-up time, which is typical in client applications.

Compared to HQEMU on ×86–64, the start-up time of HQEMU is worse than QEMU. Lynx emulated by HQEMU requires about 19× the start-up time of the LLBT translated one. This is because there are no hot traces at the start-up of Lynx, and HQEMU has to initialize more runtime components, such as the LLVM translator and the dynamic binary optimizer, than does QEMU during its start-up. In addition, although dynamic optimizations could improve the code quality for frequently executed code, they are not free. In particular, their initial cost of profiling and learning would not benefit applications that require fast start-up time. In embedded systems, many mobile applications require fast start-up. DBT often takes advantage of hot spots to amortize translation and optimization time. However, the lack of hot spots in the execution paths makes SBT a more attractive solution when faster start-up is a major concern.

4.5. Space Overhead from Address Mapping Tables

A naïve implementation of the runtime AMT could significantly increase the memory requirement of SBT-generated code. In the worst case, every ARM instruction could be the target of an indirect branch that would require one entry in the AMT. In LLBT, we pay extra attention to the AMT design to minimize the memory usage. LLBT generates a smaller table if the input ARM binaries contain symbol table information since we can explicitly know the exact function entry points. Otherwise, if the symbol tables of the input binaries are stripped off, LLBT has to include all possible addresses that might be function entry points in the AMT. With the help of the symbol table information, only 7.5% of instruction addresses on average are actually included in the AMT on the EEMBC benchmark suite. However, when no symbol tables are available, our heuristic, which is described in Section 2.4, will add only 2–7% more instruction addresses to the AMT.

4.6. Code Size Measurement and Memory Overhead

Figure 15 shows the ratio of static code size of the statically translated EEMBC suite to the native binary. The average code size, including text and data, of translated binaries is only about 1.5× of the source binaries. On some benchmarks, such as rgbcmy01, rgbhpg01, and rgbyiq01, the small size increases are because more than 98% of code size is data, and the translated instructions increased only a little compared to the data. In contrast, DBT systems require additional space for their own code and data. For example, the static code size of QEMU user mode emulator (i.e., qemu-arm) in version 1.2.2 on ×86 is about 1.2Mb (dynamically linked). However, in the EEMBC benchmark suite, the average code size of the source ARM binaries is about 21Kb, and the average code size of the translated ×86 binaries is about 32Kb.

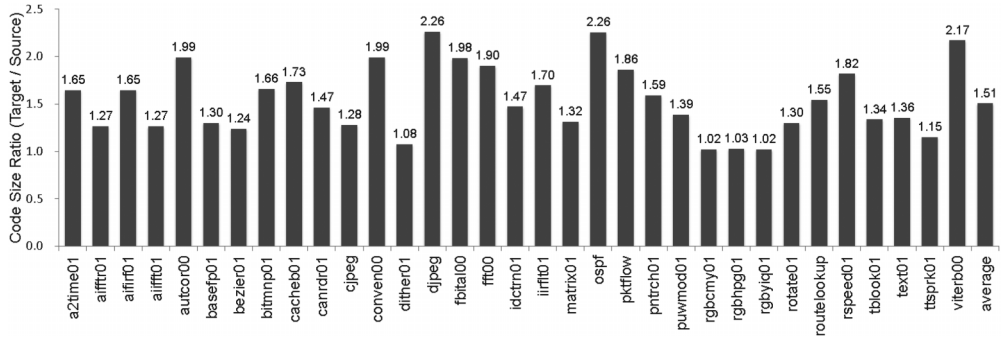

Fig. 15. Static code size ratio of "target binaries size/source binaries size" on translating the EEMBC benchmark suite from ARM to ×86. The code size includes text and data in the binaries, which are dynamically linked and stripped.

Considering the code size of shared libraries, LLBT requires a runtime library, whereas QEMU requires source (i.e., ARM) libraries to execute dynamically linked source binaries. The code size of the LLBT runtime library on ×86 is about 17 Kbytes. The code size of required ARM libraries, including C and math libraries, for running EEMBC in μ`Clibc`, which is a small C library for embedded Linux, is about 268 Kbytes. LLBT-generated code does not require source libraries because it can dynamically link with native libraries.

Furthermore, if we compare the runtime memory overhead on running the EEMBC benchmark suite, the maximum resident set size of LLBT-generated code is increased by only about 6% over native runs. On the other hand, the maximum resident set size of running on QEMU is increased by more than 400% over native runs. This is because DBT systems require additional runtime space for themselves and have to manage additional data structures (e.g., the code cache). Therefore, the runtime memory requirement for applications with SBT is lower than with DBT-based systems running the EEMBC benchmark suite. Although SBT-based systems have to translate all source instructions into target instructions, whereas a DBT system translates only portions that are actually executed, this may not be an alarming issue in embedded environments since embedded applications are small in size when compared to the space requirement of a DBT system.

In addition, we also compared the runtime memory overhead on running the SPEC CPU2006 integer benchmarks. The results show that the percentage of the increased memory overheads for QEMU and LLBT are almost the same. This is because the memory requirements for SPEC benchmarks are much larger than that of EEMBC. Therefore, the memory overheads that come from QEMU and LLBT can be ignored. However, it is rare that an embedded application will require much memory, like the SPEC benchmarks, which usually run on general-purpose computing environments.

4.7. Translation Time

We measured the translation time of LLBT on translating the EEMBC benchmark suite from ARM into ×86. The experiment was run on an Intel Xeon E5506 processor. The total translation time of each benchmark is from 0.3 to 11.2 seconds. The translation spends about 10% of time on the LLBT translator, 70% of time on LLVM optimizations and compilation, and the rest on the target assembler and linker. Although the translation time depends on the size of source binaries and would be increased on translating larger benchmarks, it is less an issue because the translation time in SBT is not part of execution time, as it is in DBT.

5. RELATED WORK

We first provide related work on SBT and DBT systems in Sections 5.1 and 5.2. Then we describe previous research related to embedded SBT and DBT systems in Sections 5.3 and 5.4, respectively.

5.1. Static Binary Translators

FX!32 [Chernoff et al. 1998] is a dynamic profiling and static binary translation system that translates ×86 Win32 applications for Windows NT/Alpha platforms. Initially, ×86 applications will be interpreted to gather execution profiles. The collected profiles will be used to guide a static binary translator to generate native Alpha code. In subsequent runs of the application, previously generated Alpha native code could be used. The interpreter in FX!32 plays an important role in discovering code, as well as in handling possible SMC issues. UQBT [Cifuentes and Emmerik 2000; Cifuentes et al. 2002b] is a retargetable SBT tool that translates source binaries into a high-level intermediate language called HRTL. Then, HRTL is further translated into various forms depending on the translation's purpose.

DisIRer [Hwang et al. 2010] leverages the GCC infrastructure to build a multi-platform static binary translator. DisIRer uses a pattern matching mechanism that translates ×86 instruction sequences into GCC's low-level Register Transfer Language (RTL) representation and then translates RTL into GCC's high-level Abstract Syntax Tree (AST). Thereafter, DisIRer can rely on GCC's optimizer and code generator for generating code for different target machines. The developers of DisIRer did not discuss their work on how to handle the code discovery and code location problems, which are particularly important for ×86 architecture.

Bansal and Aiken [Bansal and Aiken 2008] applied peephole superoptimization to binary translation. They used a peephole superoptimizer on an SBT to perform efficient binary translation. Their approach is similar to automatic construction of peephole optimizers using superoptimization [Bansal and Aiken 2006]. Their results show good performance compared with natively compiled executables. However, if source binaries are not optimized, the performance improvement may be decreased because the peephole optimization only performs over short instruction sequences. Superoptimization is performed via exhaustive search, and the search space grows exponentially with the peephole size. In contrast, LLBT performs LLVM optimizations and global register allocation on LLVM IR to generate optimized code.

5.2. Dynamic Binary Translators

Although there are many fast dynamic binary translators, they are usually machine specific [Baraz et al. 2003; Ebcioğlu and Altman 1997; Zheng and Thompson 2000]. QEMU [Bellard 2005] is a fast emulator that adopts retargetable DBT techniques. The current versions of QEMU use Tiny Code Generator (TCG) to turn source binaries into IRs and translate IRs into different target native code. UQDBT [Ung and Cifuentes 2000] and Walkabout [Cifuentes et al. 2002a] also support retargetability through the use of specifications of the syntax and semantics of target machine instructions. Strata [Scott et al. 2003] is another retargetable BT that defines a reconfigurable target interface. In Strata, only the target-specific functions required by the target interface need to be implemented when retargeting to a new platform.

Some DBT systems focused on minimizing the dynamic translation and runtime optimization overhead. HP's Aries [Zheng and Thompson 2000], IBM's BOA [Gschwind et al. 2000], and Walkabout [Cifuentes et al. 2002a] combined an interpreter with their DBT systems to collect a runtime profile and limit the scope of dynamic translation to a block or a trace of instructions. This requires additional interpretation before

dynamic translation to identify frequently executed code. Similarly, IA-32 Execution Layer [Baraz et al. 2003] adopted a two-phase translation design that translates cold code first with minimal optimizations and uses instrumentations to gather profiles for later optimizations on hot code. HQEMU [Hong et al. 2012] is a trace-based DBT system that combines a fast translator (i.e., QEMU's TCG) with an optimized translator based on LLVM. It takes advantage of multicore processors to translate/optimize hot traces on another thread. However, such DBT systems may not be suitable for short-running or interactive applications, which require fast start-up and response time.

5.3. Static Binary Translators for Embedded Systems

Chen et al. proposed an SBT system [Chen et al. 2008] that migrates executables from ARM to an MIPS-like architecture and applies some optimizations. Unlike LLBT, this is a direct ISA-to-ISA SBT. It exhibits more optimization opportunities for special cases. However, a direct ISA-to-ISA translation lacks retargetability. For instance, it took two weeks to port the ARM-to-×86 LLBT to ARM-to-MIPS LLBT. Furthermore, LLBT leverages rich machine-independent and machine-dependent optimizations developed in LLVM. It is very costly to develop, test, and verify such optimizations for a direct ISA-to-ISA binary translator.

De Sutter et al. proposed a link-time optimizer that performs static binary rewriting at link time for ARM executables [De Sutter et al. 2007; Chanet et al. 2007]. Although their link-time optimizer is not a BT system, their work on code discovery is closely related to LLBT. Like the link-time optimizer, LLBT also locates compiler-generated code (i.e., jump tables) based on pattern recognition. However, code discovery in a link-time optimizer for ARM is much easier than SBT because it can utilize symbol information to identify ARM instruction regions, Thumb instruction regions, and data regions. Moreover, they can handle hand-crafted assembly by patching compilers to emit symbols that identify where the manually written assembler code is located [Chanet et al. 2007]. However, this approach is not practicable for BT systems because the source code may not be available.

5.4. Dynamic Binary Translators for Embedded Systems

Some recent research focused on improving DBT's memory footprint and performance for embedded systems. Guha et al. [2007] proposed techniques to reduce the size of exit stubs in the translated code. They also proposed a dynamic trace selection strategy and a selective cache flushing strategy to improve memory efficiency and performance [Guha et al. 2012]. Baiocchi et al. [2008] proposed techniques to minimize the space needed by trampolines, indirect branch handling, and context switch code in a DBT for embedded systems with scratchpad memory. Their approach can significantly improve performance in constrained code caches. They also developed methods to compress the evicted code in constrained code caches to avoid re-translating previously seen code [Baiocchi et al. 2007]. Moore et al. [2009] provided another form of code cache management on embedded systems. Their techniques improve locality and reduce TLB pressure by arranging data and code in code caches for the ARM architecture. However, these DBTs' methods address the management of constrained code caches and focus on utilizing code caches effectively. LLBT tries to avoid runtime translation overhead and improve the quality of generated code. There are no code cache issues for SBT.

6. CONCLUSIONS

This paper presents a retargetable SBT called LLBT, which translates ARM binaries into LLVM IRs and leverages the LLVM to generate binaries for different target architectures. DBT is usually the first choice for migrating application binaries from one ISA to another. However, for embedded systems, start-up time, memory usage, and power

efficiency are important considerations. SBT can be far more efficient than DBT when migrating short-running applications. In the past, SBT has been downplayed due to the challenges in code discovery, code location, and issues like SMC. To migrate ARM-based binaries to other ISAs, we have devised various techniques to effectively handle the code discovery problem such as ARM/Thumb region identification, PC-relative data inlining, and jump table recovery. To handle indirect branches, a large AMT is used in SBT. Because we target compiler-generated code rather than hand-crafted binaries, we could narrow down the possible destinations of indirect branches for inclusion in the AMT. In addition, SMC rarely happens in the applications available in Google Play. Hence, we believe using SBT for migrating applications under Android for embedded systems to be a viable and practical solution.

In this article, we have discussed issues related to code generation in terms of register mapping, conditional execution, indirect branch, and helper function replacement. These are techniques that make LLBT an effective and efficient retargetable binary translator. Our experiments show that LLBT-translated code can run more than 6× and 2.3× faster than a retargetable DBT system (i.e., QEMU) and a trace-based DBT system (i.e., HQEMU), respectively, on small and short-running embedded benchmarks. With the help of the comprehensive optimizations in LLVM, LLBT can effectively support migrating applications to new architectures, especially embedded environments. We have demonstrated its retargetability by translating ARM binaries into ×86, ×86–64, ARM, and MIPS on real hardware, with high migration efficiency.

REFERENCES

E. R. Altman, D. Kaeli, and Y. Sheffer. 2000. Welcome to the opportunities of binary translation. *Computer* 33, 3 (March 2000), 40–45.

Kristy Andrews and Duane Sand. 1992. Migrating a CISC computer family onto RISC via object code translation. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*. ACM, New York, 213–222.

ARM Limited 2012. *Run-time ABI for the ARM Architecture*. ARM Limited.

Jose Baiocchi, Bruce R. Childers, Jack W. Davidson, Jason D. Hiser, and Jonathan Misurda. 2007. Fragment cache management for dynamic binary translators in embedded systems with scratchpad. In *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'07)*. ACM, New York, 75–84. DOI:http://dx.doi.org/10.1145/1289881.1289898

José A. Baiocchi, Bruce R. Childers, Jack W. Davidson, and Jason D. Hiser. 2008. Reducing pressure in bounded DBT code caches. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'08)*. ACM, New York, 109–118. DOI:http://dx.doi.org/10.1145/1450095.1450114

Sorav Bansal and Alex Aiken. 2006. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*. ACM, New York, 394–403. DOI:http://dx.doi.org/10.1145/1168857.1168906

Sorav Bansal and Alex Aiken. 2008. Binary translation using peephole superoptimizers. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, 177–192.

Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigel Zemach. 2003. IA-32 execution layer: A two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'36)*. IEEE Computer Society, Washington, DC, 191–204.

Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC'05)*. USENIX Association, Berkeley, CA, 41–41.

Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*. IEEE Computer Society, Washington, DC, 265–275.

Dominique Chanet, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere. 2007. Automated reduction of the memory footprint of the Linux kernel. *ACM Transactions in Embedded Computer Systems* 6, 4, Article 23 (Sept. 2007). DOI:http://dx.doi.org/10.1145/1274858.1274861

Jiunn-Yeu Chen, Bor-Yeh Shen, Yuan-Jia Li, Wuu Yang, and Wei-Chung Hsu. 2013a. Automatic validation for static binary translation. In *Proceedings of the 2nd Asia-Pacific Programming Languages and Compilers Workshop (APPLC'13)*. ACM, New York, 1–9.

Jiunn-Yeu Chen, Bor-Yeh Shen, Quan-Huei Ou, Wuu Yang, and Wei-Chung Hsu. 2013b. Effective code discovery for ARM/Thumb mixed ISA binaries in a static binary translator. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'13)*. ACM, New York, IEEE, Piscataway, NJ, 1–10.

Jiunn-Yeu Chen, Wuu Yang, Charlie Su, and Wei-Chung Hsu. 2008. A static binary translator for efficient migration of ARM based applications. In *Proceedings of the 6th Workshop on Optimizations for DSP and Embedded Systems (ODES'08)*. ACM, New York, 55–64.

Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. 1998. FX!32: A profile-directed binary translator. *IEEE Micro* 18, Issue 2 (March 1998), 56–64.

Cristina Cifuentes and Mike Van Emmerik. 1999. Recovery of jump table case statements from binary code. In *Proceedings of the 7th International Workshop on Program Comprehension (IWPC'99)*. IEEE Computer Society, Washington, DC, 192–199.

Cristina Cifuentes and Mike Van Emmerik. 2000. UQBT: Adaptable binary translation at low cost. *Computer* 33 (March 2000), 60–66. Issue 3.

Cristina Cifuentes, Brian Lewis, and David Ung. 2002a. *Walkabout: A Retargetable Dynamic Binary Translation Framework*. Technical Report. Mountain View, CA.

Cristina Cifuentes and Vishv M. Malhotra. 1996. Binary translation: Static, dynamic, retargetable? In *Proceedings of the 1996 International Conference on Software Maintenance (ICSM'96)*. IEEE Computer Society, Washington, DC, 340–349.

Cristina Cifuentes, Mike Van Emmerik, Norman Ramsey, and Brian Lewis. 2002b. *Experience in the Design, Implementation and Use of a Retargetable Static Binary Translation Framework*. Technical Report. Mountain View, CA.

Bjorn De Sutter, Ludo Van Put, Dominique Chanet, Bruno De Bus, and Koen De Bosschere. 2007. Link-time compaction and optimization of ARM executables. *ACM Transactions on Embedded Computer Systems* 6, 1 (Feb. 2007), Article 5. DOI:http://dx.doi.org/10.1145/1210268.1210273

Kemal Ebcioğlu and Erik R. Altman. 1997. DAISY: Dynamic compilation for 100 compatibility. *SIGARCH Comput. Archit. News* 25, 2 (May 1997), 26–37. DOI:http://dx.doi.org/10.1145/384286.264126

Michael Gschwind, Erik R. Altman, Sumedh Sathaye, Paul Ledak, and David Appenzeller. 2000. Dynamic and transparent binary translation. *Computer* 33, 3 (March 2000), 54–59.

Apala Guha, Kim Hazelwood, and Mary Lou Soffa. 2007. Reducing exit stub memory consumption in code caches. In *Proceedings of the 2nd International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'07)*. Springer-Verlag, Berlin, 87–101.

Apala Guha, Kim Hazelwood, and Mary Lou Soffa. 2012. Memory optimization of dynamic binary translators for embedded systems. *ACM Transactions on Architecture and Code Optimization* 9, 3, Article 22 (Oct. 2012), 29 pages. DOI:http://dx.doi.org/10.1145/2355585.2355595

Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. 2012. HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO'12)*. ACM, New York, 104–113. DOI:http://dx.doi.org/10.1145/2259016.2259030

Yuan-Shin Hwang, Tzong-Yen Lin, and Rong-Guey Chang. 2010. DisIRer: Converting a retargetable compiler into a multiplatform binary translator. *ACM Transactions on Architecture and Code Optimization* 7, 4 (December 2010), Article 18, 36 pages.

Dawid Kurzyniec and Vaidy Sunderam. 2001. Efficient cooperation between Java and native codes - JNI performance benchmark. In *Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*. Computer Science Research, Education, and Applications Press, Athens, GA, 2232–2239.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. IEEE Computer Society, Washington, DC, 75–86.

Sangchul Lee and Jae Wook Jeon. 2010. Evaluating performance of Android platform using native C for embedded systems. In *Proceedings of the 2010 International Conference on Control Automation and Systems (ICCAS'10)*. IEEE Computer Society, Washington, DC, 1160–1163.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with

dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, New York, 190–200.

Ryan W. Moore, José A. Baiocchi, Bruce R. Childers, Jack W. Davidson, and Jason D. Hiser. 2009. Addressing the challenges of DBT for the ARM architecture. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'09)*. ACM, New York, 147–156. DOI:http://dx.doi.org/10.1145/1542452.1542472

Steven S. Muchnick. 1998. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

Kevin Scott, Jack W. Davidson, and Kevin Skadron. 2001. *Low-Overhead Software Dynamic Translation*. Technical Report CS-2001-18. Department of Computer Science, University of Virginia, Charlottesville.

K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. 2003. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*. IEEE Computer Society, Washington, DC, 36–47.

David Seal. 2000. *ARM Architecture Reference Manual*. Addison-Wesley Longman.

Bor-Yeh Shen, Jyun-Yan You, Wuu Yang, and Wei-Chung Hsu. 2012. An LLVM-based hybrid binary translation system. In *Proceedings of the 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*. IEEE Computer Society, Washington, DC, 229–236.

Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. 1993. Binary translation. *Communication ACM* 36, 2 (February 1993), 69–81.

Jim Smith and Ravi Nair. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann.

David Ung and Cristina Cifuentes. 2000. Machine-adaptable dynamic binary translation. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO'00)*. ACM, New York, 41–51.

Chih-Sheng Wang, Guillermo Perez, Yeh-Ching Chung, Wei-Chung Hsu, Wei-Kuan Shih, and Hong-Rong Hsu. 2011. A method-based ahead-of-time compiler for android applications. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'11)*. ACM, New York, 15–24. DOI:http://dx.doi.org/10.1145/2038698.2038704

Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. 2008. STILL: Exploit code detection via static taint and initialization analyses. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC'08)*. IEEE, Los Alamitos, CA, 289–298.

Cindy Zheng and Carol Thompson. 2000. PA-RISC to IA-64: Transparent execution, no recompilation. *Computer* 33, 3 (March 2000), 47–52.

Received October 2013; revised January 2014; accepted January 2014