An LLVM-based Hybrid Binary Translation System

Bor-Yeh Shen, Jyun-Yan You, Wuu Yang, Wei-Chung Hsu
Institute of Computer Science and Engineering
National Chiao Tung University
Hsinchu City 30010, Taiwan, R.O.C.
{byshen,jyyou,wuuyang,hsu}@cs.nctu.edu.tw

***Abstract*—Static binary translation (SBT) systems and dynamic binary translation (DBT) systems have their own merits and disadvantages. SBT can perform whole-program optimizations and do not incur run-time overheads. However, the code discovery and the code location problems caused by indirect branches make SBT systems hard to develop. On the other hand, DBT can perform optimizations based on program's runtime behaviors and can handle indirect branches easily. However, because the translation time accounts for a part of the execution time, DBT systems cannot perform aggressive optimizations. Therefore, quality of the code generated by DBT is not as good as that by SBT. In this paper, we present a hybrid binary translation (HBT) system which combines the merits of both SBT and DBT. It leverages the LLVM infrastructure to translate source binary code, optimize, and generate target binary code. It first translates binary statically. If a run-time exception happens, the HBT system switches to dynamic translation. On the EEMBC benchmark suite, our experimental result shows that the HBT system can run about 4 to 20 times faster than a LLVM-based DBT system.**

I. Introduction

Binary translation [1], [2] techniques have been actively studied and developed for the past two decades. They have been widely adopted in many different areas such as fast simulation, software security enforcement, application profiling [3], and system virtual machine implementation [4], and have become a standard approach for migrating applications from one ISA to another. For example, Apple developed Rosetta [5], which dynamically translates PowerPC programs into Intel X86 binaries, Hewlett-Packard developed Aries [6] to dynamically translate PA-RISC executables to IA-64 architecture, and DEC developed FX!32 [7] to make X86 Win32 applications runnable on Windows NT/Alpha platforms.

Software-based binary translation systems can be roughly classified in two categories: static binary translation (SBT) and dynamic binary translation (DBT) [8], [9]. Nowadays, most application migrations are based on DBT systems because both the code discovery and the code location problems [4] can be handled more efficiently by DBT techniques. Recent research is also focused on dynamic binary translation rather than static binary translation. However, DBT has its own limitations, such as the overhead of run-time translation, lack of code optimizations for translated binaries, and increased memory usage. Although such shortcomings may not be serious issues for general purpose computing environments, they are unacceptable for embedded computing where the application start-up time, the power consumption, and memory size are primary concerns.

On the other hand, code discovery and the code location are the well known problems of SBT. The code discovery problem refers to the difficulty to determine the target of an indirect jump statically since the content of the jump register is not known at compile time, and the difficulty of distinguish instructions from data if the programmer intersperse data with code. The code location problem refers to the handling of indirect jumps where the jump target address must be mapped to the address of the translated code. Both problems are particularly challenging with CISC machines where instructions have variable lengths. However, such problems can be less complex when dealing with fixed-length instructions and many embedded processors, such as ARM, MIPS, PowerPC, and SuperH, are RISC ISAs.

To mix the advantages of both SBT and DBT systems, we propose a hybrid binary translation (HBT) system that translates binary code statically and links the translated binary with a run-time dynamic binary translation system. When a run-time exception occurs, such as failing to find a branch address, the program switches to the run-time system, which translates source binary dynamically. After dynamic translation and execution, it may continue dynamic translation or may switch back to the static translated binary, depending on the search results of the address translation table.

Besides, development cost is one of the important concern in developing binary translation systems. Crafting a binary translator from scratch requires extraordinary effort, especially in the design and verification of various optimizations. Moreover, since binary translators are highly target-machine-dependent, most of the efforts may not be leveraged if the target machine is changed, which is more common in the embedded world. Therefore, retargeting a binary translator to a different platform, especially when serious optimizations are required, has been a highly desirable but challenging task.

To come up with a retargetable binary translator, we took the general approach of portable compilers which separate the translation process into target-dependent and target-independent parts, interfaced by intermediate representation (IR) forms. The front-end of our HBT system would translate the executable of one machine into IR forms and then the backend translates the IR forms into binaries of a machine with different ISA. However, instead of designing a new target-independent IR and implementing a code generator for various architecture, we tended to leverage an existing compiler infras-

978-1-4673-2684-1/12/$31.00 ©2012 IEEE

tructure to build a high-performance and retargetable binary translator.

In this paper, we present a HBT system which leverages the LLVM [10] infrastructure and can translate ARM-based binaries to many different platforms supported by LLVM.

The remainder of this paper is organized as follows: section II lists some related work. Section III gives an overview of our translator. In section IV we describe the implementation details. Section V discusses the experimental results and section VI concludes the paper.

II. Related Work

This section will introduce related work of static and dynamic binary translation systems, and give an overview of the LLVM compiler infrastructure.

A. Static binary translator

UQBT [11], [12] is a retargetable SBT tool which translates binaries into a high-level intermediate language, HRTL, then HRTL will be translated into various forms depending on the translation purpose. For example, the forms can be low-level C code, object code, and VPO RTL. However, UQBT still relies on a run-time interpreter for handling indirect register calls that can not be discovered at static time. Compared with UQBT, our translator uses LLVM intermediate language to achieve retargetability, and use a dynamic translator instead of an interpreter for handling exceptions.

Chen et al. [13] built a SBT system that translates ARM binaries directly into a MIPS-like platform without using target-independent IR. Their translator can efficiently migrate ARM applications without using a run-time emulator if the ARM binaries are generated by the GCC compiler. However, if the binaries contain handcrafted assembly, the translated code may fail to work because it may contain some indirect branch targets that can not be found by their translator.

B. Dynamic binary translator

QEMU is a fast and portable dynamic binary translator, which supports both user-mode and full-system emulations. The initial QEMU implementation translates source binary instructions into a sequence of micro operations, and such micro operations are implemented by a small piece of C code and the C code is pre-compiled by GCC into target binary. At runtime, each micro operation is replaced by the pre-compiled binary. Newer versions of QEMU switched to use IR (Intermediate Representations) in their new code generator called TCG (Tiny Code Generator). However, QEMU does not optimize their generated code. Some LLVM-based dynamic binary translators [14], [15], [16], [17] utilize existing emulation tools. They replace the code-generation layer with LLVM components and let the emulator do the remaining jobs. Compared with them, our translator is not merely using LLVM to optimize translated code, but combining the ability of static binary translation to do more aggressive optimizations at static time.

Some DBT systems focused on minimizing the dynamic translation and run-time optimization overheads. HP's Aries [6], IBM's BOA [18], and Walkabout [19] combined an interpreter with their DBT systems to collect run-time profile information and restrict the scope of dynamic translation to a short hot path of instructions. This requires additional interpretation before dynamic translation to find the frequently executed code. Similarly, IA-32 Execution Layer [20] adopted a two-phase translation design which translates code first with minimal optimizations and uses instrumentation to identify hot code. However, these DBT systems may result in full interpretation or poor code quality in embedded systems since most embedded applications are client-side programs, which have only short execution time. Compared with them, our HBT system does dynamic translation only when exceptions happen. Most of time it executes statically translated code with small overheads.

C. LLVM

LLVM is an open source compiler framework which can convert machine-independent instructions to machine-dependent assembly code. In addition to a static compiler, it includes a just-in-time (JIT) compiler and Machine Code (MC) [21] toolkit as the parts of the suit. The MC toolkit provides tools for CPU instruction set related work, such as assembler, disassembler, and object file format handling, etc. With the rapid development in recent years, there have been various advanced optimizations and analysis phases added to the LLVM infrastructure. The relative robust infrastructure and the rich optimizations, features available in LLVM has inspired us to build a portable binary translator on top of LLVM.

III. Overview

Binary translation translates binary code from one architecture to another. There are two types of binary translation – *static* binary translation (SBT) and *dynamic* binary translation (DBT). The former translates binary code before the program starts execution while the latter translates binary code at run time. Most current binary translators adopt the dynamic approach because it can handle the code discovery and the code location problems more easily. However, DBT systems cannot perform aggressive optimizations, which may incur severe run-time overheads. In contrast, SBT systems can perform time-consuming, deep optimizations. Nevertheless, SBT needs a mechanism to handle indirect branches and data that are embedded in code segment.

Some SBT systems [13] which can find all possible indirect branch targets if the binaries are generated by specific compilers, such as GCC. However, it will be hard to find all destinations of indirect branch instructions in handcrafted assembly, which can usually be found in system libraries. Conceptually, it is possible to map all addresses in the code segment which would result in a huge (and infeasible) address mapping table. In order to solve these problems, we propose a retargetable hybrid binary translation (HBT) system which combines the merits of SBT and DBT systems.

Figure 1 shows the flow of static translation. Source binaries can be statically or dynamically linked. Our translator use

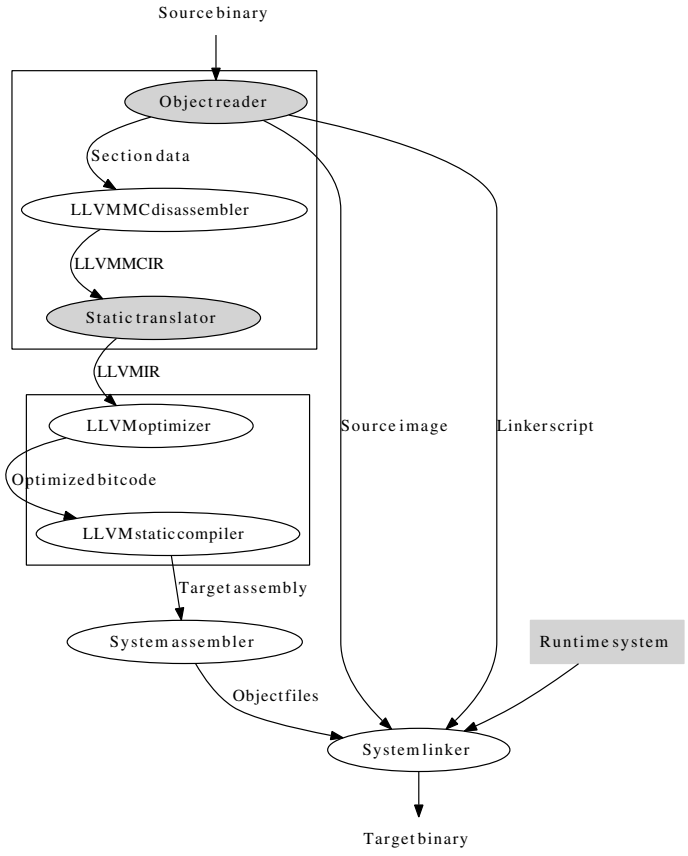

Fig. 1. Static translation

LLVM API to read a source binary file, disassemble instructions and translate them into LLVM instructions. Below is a summary of the static translation.

1) Instructions in the source binary are read and decoded by the object reader and the LLVM MC disassembler respectively.
2) The static translator translates the IR of the LLVM MC disassembler into LLVM instructions (LLVM IR) and saves the result to an LLVM bitcode file.
3) The LLVM optimizer (`opt`) is used to perform target-independent optimizations on the above bitcode.
4) The optimized bitcode is compiled to the target assembly by the LLVM static compiler (`llc`). When compiling bitcode to target assembly, `llc` also performs some target-specific optimizations.
5) The target assembly is assembled to target object code by the target assembler.
6) Finally, the target linker reads a customized linker script generated by the object reader and links the source image together with the target objects. A run-time system which includes a dynamic translator and a system call emulator is dynamically linked with the target binary.

The flow of execution, and dynamic translation, shown in Figure 2, consists of two parts. The first part is to execute statically translated code. The second part is dynamic translation. When the destination of an (indirect) branch instruction cannot be found (in the address mapping table) during execution, our HBT system switches to dynamic translation. Below is a summary of dynamic translation in HBT.

1) Before the translated binary starts execution, it needs initialization, which includes creating a dynamic translator, allocating stack space for the translated binary, pushing arguments into the source stack, etc.
2) After initialization, execution starts from the program entry point.
3) If the instruction under execution is an indirect branch, the address mapping table is consulted to find the corresponding address in the target binary. Execution continues from that address.
4) If the address mapping table does not contain an entry for the destination of the indirect branch instruction, it will switch to the dynamic translator and looks up the destination of the indirect branch instruction in the code cache. If found, the translated code in the code cache is executed directly. Otherwise, if the destination has not been translated yet, The dynamic translator will find the destination in the source image, translate source instructions, put the translated code in the code cache, and execute the translated code.

Figure 3 shows the flow of dynamic translation. It is similar to the static translator in Figure 1. Below is the summary of the dynamic translation process.

1) The dynamic translator translates a contiguous block of instructions (in the source image) into LLVM IR until it meets a branch instruction and performs some basic optimizations on these IR.
2) The LLVM IR is compiled to native code by the LLVM JIT compiler. Some target-specific optimizations are also performed in this stage.
3) The source address of block entry and the entry of its corresponding native code are saved to an address translation table for later code cache lookup.

IV. Implementation

We implemented a hybrid binary translator which can translate ARM binaries into X86 binaries. Figure 4 shows the memory layout of the target binary. For the performance reason, the text, data, and BSS sections of the original ARM binary are copied to the target binary at the same address. Therefore, we do not have to do the address remapping for memory access while program execution. In this section, we will explain the implementation details of our hybrid translator.

A. Hybrid Translation

The dynamic translator is implemented as a library and is linked with target binaries. It translates source instructions one by one until a branch instruction is encountered. The translated instructions are placed in an LLVM function and is later translated into a native function. The last translated instruction, which is a branch instruction, is translated into a

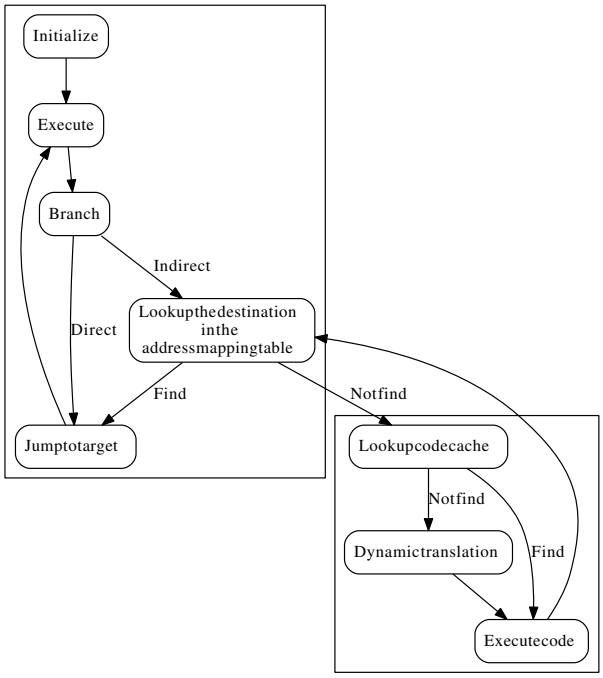

Fig. 2. Execution flow

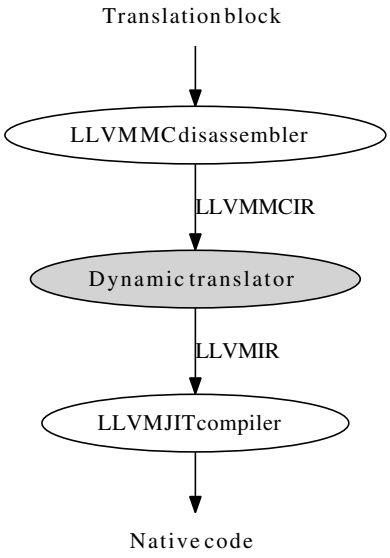

Fig. 3. Dynamic translation

`return` instruction, which returns the destination address of the branch. The destination points to the next instruction for execution. Figure 5 shows a translated LLVM function. We may notice that last basic block does not return an address directly. Instead, it returns the return value of an unnamed function. This is for doing block chaining which is a common optimization in dynamic binary translation. We will discuss the details in section IV-E.

When the target binary starts execution, it first initializes the

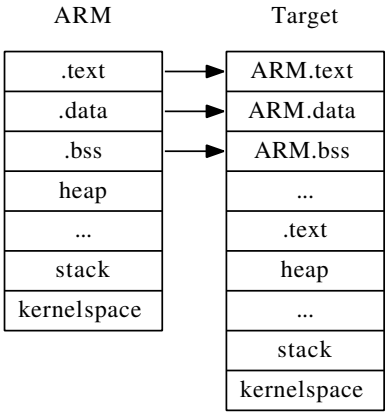

Fig. 4. Memory layout of target binary

```
define internal fastcc i32 @L_00009be8_() {
entry:
......

L_00009c20_34:
; preds = %L_00009c20_
  store i32 -1, i32* @R0, align 4
  br label %L_00009c24_
......

L_00009c28_:
......
  store i32 %92, i32* @PC, align 4
  %94 = tail call fastcc i32 @75(i32 %92)
  ret i32 %94
}
```

Fig. 5. A dynamically translated block (an LLVM function)

dynamic translator and allocates memory space for emulating the call stack of source binaries. In order to handle command-line arguments and environment variables, it copies the argument count, argument vector, and environment variables from the target stack to the source stack. If the source binary is dynamically linked, it also loads the source dynamic linker into memory and pushes the auxiliary vector, which transfers OS-specific information to the program, into the source stack. After initialization, the target binary jumps to its program entry point and starts execution.

B. Indirect Branch Handling

For some indirect branch instructions whose destinations cannot be determined during static time, we use an *address mapping table* or dynamic translation to resolve the destinations. The address mapping table maps an address in the source image to the corresponding address in the target binary. In order to reduce the size of the address mapping table, only

plausible, but not all, addresses are kept in the table. If a branch target address cannot be found in the address mapping table, the dynamic translator is invoked that translates a block of code in the source image into a block of code for the target platform. The generated binary code, which has the form of a native function, is stored in the code cache. After translation and executing the block of code, the next branch address is returned. Again the address mapping table is consulted to locate the returned address. If it is in the table, execution is restarted from the target binary. If not, the dynamic translator repeats the above translation process.

We implemented the address mapping table by using the LLVM `switch` instruction. Typically, compilers will generate a jump table for improving the performance of switch statements. However, since our translator only collect a few possible indirect branch addresses from source binaries, if we put the addresses in an LLVM switch instruction, it will be too sparse to generate a jump table. Instead, compilers will generate a sequence of if-else instructions for the switch instruction. This will result in bad performance for searching an address in the address mapping table. In order to solve the problem, we use a simple hash function to split a large switch statement into several small switch statements.

Figure 6 shows an example of the address mapping table. The number of tables is dependent on how many possible indirect branch targets we found at static time. In this example, the number of tables is 16. Figure 7 shows the address mapping table in LLVM IR. <label>:629 is on the first level and labels from <label>:683 to <label>:698 are on the second level. It loads an address from PC, which is the program counter in the source binary, calculates a hash value and looks up the table. If the address is not in the address mapping table, execution branches to <label>:633 where the dynamic translator is invoked by calling `transInst` function. After using the two-level switch approach, the hash values in the first level switch become dense enough to generate as a jump table. The values of second level switch statements are still sparse. However, the size of switch statements in the second level becomes small and takes less time to search an address.

C. Register Mapping

In our translator, each register of the source ISA is mapped to an LLVM global variable. In order to let the LLVM compiler promote them to target registers, we have to restrict the usage of these variables only within the statically-translated binary. However, if we want to share these registers between the statically-translated binary and the native code generated by the dynamic translator, the variables needs to be visible by the dynamic translator. As a result, we use two sets of global variables for simulating ARM registers. One is for the statically-translated binary only, the other is for the dynamic translator only. When switching between the dynamic translator and the statically-translated binary, a context switching between these two sets of variables needs to be performed. In figure 7, <label>:633 updates the variables before and after switching to the dynamic translator.

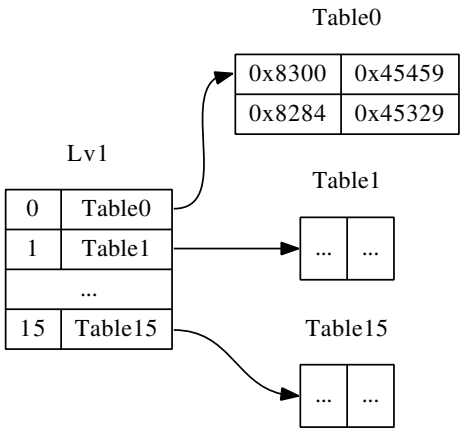

Fig. 6. Diagram of Address Mapping Table

In addition to the overhead of context switching, dynamic translation also incurs run-time overheads. These include disassembling source instructions, translating MC IR to LLVM IR, and compiling LLVM IR to the dynamically translated blocks. In order to reduce the frequency of switching to the dynamic translator, some optimizations in the static and dynamic translators are implemented. We will discuss them in section IV-D and section IV-E.

D. Finding Indirect Branch Addresses

The easiest way to avoid switching to the dynamic translator is to include all source instruction addresses in the address mapping table. This approach will create a table of intolerable size. For example, a 4 bytes ARM instruction needs 4 bytes for saving its address and another 4 bytes for saving its corresponding address of the translated code in the address mapping table. It will spend twice memory space than the original ARM instruction.

In order to keep the size of the address mapping table as small as possible, our static translator only collects return addresses, function pointers, and function entry points in the table. In our experimental result, this table is very effective and the dynamic translator is seldom invoked

- Return address is the address of the instruction that a function returns to. It is the address of the instruction that immediately follows a function call instruction.
- A function pointer is stored as a piece of PC-relative data in the ARM binary. We examine all the PC-relative load instructions in the ARM binary. If the loaded data is an address located at the code segment, it is included in the address mapping table.
- Function entry points can be found in the symbol table in the ARM binary. If the symbol table is stripped off, the next instruction that immediately follows a `return` instruction is considered as a function entry and its address will be included in the address mapping table. If there is data immediately follows a `return` instruction, it will

```
L_AddressMappingStub:
  %628 = load i32* @PC
  br label %629

; <label>:629
  %630 = phi i32 [ %628, %L_AddressMappingStub ],
                 [ %658, %633 ]
  %631 = lshr i32 %630, 3
  %632 = and i32 %631, 15
  switch i32 %632, label %633 [
    i32 0, label %683
    i32 1, label %684
    ......
    i32 14, label %697
    i32 15, label %698
  ]

; <label>:633
  %634 = load i32* @R9
  store i32 %634, i32* @ext_R9
  ......
  %658 = call i32 @transInst(i32 %630)
  ......
  %682 = load i32* @ext_PC
  store i32 %682, i32* @PC
  br label %629

; <label>:683
  switch i32 %630, label %633 [
    i32 33536, label %L_00008300_
    i32 33412, label %L_00008284_
  ]
  ......

; <label>:698
  switch i32 %630, label %633 [
  ]
```

Fig. 7. Address Mapping Table

be skipped and the next data is considered recursively. Besides, we also look at call instructions to find function entries.

For indirect branch targets of C++ virtual methods, their addresses are often stored in data and rodata (i.e., read only data) sections. We can look at data in these sections to find possible function entries. However, it is a rare case, especially on embedded applications. We implemented it as an option that can be used selectively.

In addition to mapping plausible addresses with the address mapping table, we also recover the `switch` statements, which makes use of indirect branches. In the ARM binary generated by GCC [22], a `switch` statement consists of a `load` instruction followed by a `compare` instruction and several

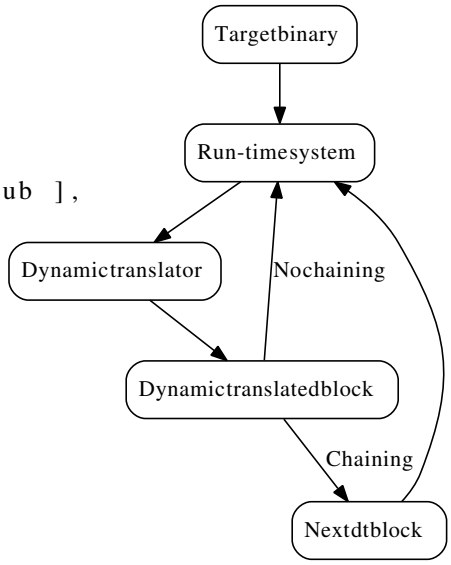

Fig. 8. Block chaining

```
define internal fastcc i32 @75(i32 %1) {
  call void asm sideeffect "", "r"(i32 -1)
  call void asm sideeffect "", "r"(i32 -2)
  call void asm sideeffect "", "r"(i32 -3)
  ret i32 %1
}
```

Fig. 9. Stub function for code in Figure 5

branch addresses. All such instruction sequences are translated into an LLVM `switch` instruction. Recovering the `switch` statements can avoid invoking the dynamic translator.

With the address mapping table and the recovered `switch` statements, the dynamic translator is invoked relatively infrequently. The time taken by the dynamic translation process only accounts for small proportion of total execution time according to our benchmark. (See section V on our experiments.)

E. Block Chaining

In dynamic translation, switching between the run-time system and the dynamically translated blocks (abbreviated as *dt blocks*) causes significant overhead. (Figure 5 is a sample dt block.) In order to reduce the switching time, the dt blocks are *chained* together if applicable. Figure 8 shows the block chaining.

Every dt block ends with a `return` instruction, whose return value is the destination address of a branch instruction. (The `return` instruction will return control to the run-time system, which will try to execute the instruction at the returned address.) If this address appears in the address mapping table, the `return` instruction can be replaced with a `branch` instruction. Hence, control will be passed directly to the next dt block, rather than to the run-time system.

In order to patch a dt block, we need to add extra instructions before the `return` instruction of the dt block.

```
mov     $0xffffffff,%eax
mov     $0xfffffffe,%eax
mov     $0xfffffffd,%eax
mov     %ecx,%eax
ret
```

Fig. 10. Native code of the stub function in Figure 9

```
cmp     $0x9360,%ecx
je      0xb65d4e00 <L_00009360_>
nopl    (%eax)
mov     %ecx,%eax
ret
```

Fig. 11. Patched stub function

However, we cannot control what target instructions will be generated by LLVM because code generation is done by an existing module in LLVM. Fortunately, we can use inline assembler expressions in LLVM IR to generate several dummy instructions. In Figure 9, the three `call` expressions are inline assembler expressions. In Figure 10, the three `mov` instructions are dummy instructions. These dummy instructions are patched when control is passed back to the run-time system. In Figure 11, the three instructions —**cmp**, **je**, and **nop**— are the patched instructions.

There is another problem. We cannot control the location where an instruction will be placed. Inline assembly is generated before the function epilogue. There may be an arbitrary number of `pop` instructions before the `return` instruction. It is hard to identify the dummy instructions.

In order to solve this problem, the dummy instructions are placed in a brand new *stub* function, which just returns the next branch address. We also replace the `return` instruction in the original dt block with a tail call to this stub function. In Figure 5, "`call i32 @75`" is a tail call. LLVM can perform the tail call optimization to generate a `branch` instruction for the tail call. Because the stub function is very simple, we can easily find the dummy instructions. Figure 9 shows the stub function for chaining the dt block of figure 5. It only returns next branch destination.

Figure 10 shows the native code of the stub function. Figure 11 shows the patched one. For the X86 target platform, the three dummy `mov` instructions are replaced with a `compare` instruction followed by a `je` instruction (which is a `branch`) and a `nop`. It compares the block entry with the next branch address. If they are the same, execution starts from the block entry. Otherwise, control is passed back to the run-time system. In the best case, it only uses two `branch` instructions to jump to the translated block instead of switching contexts and searching in the address mapping table.

V. Experimental Results

Since our HBT system can run in a pure dynamic translation mode without the static binary translator involved, in order to compare the difference between LLVM-based HBT and DBT systems, we run benchmarks on our HBT system in both the hybrid translation mode and the pure dynamic translation mode to show the performance improvement in our HBT system.

A. Environment

We use EEMBC version 1.1 [23] as our benchmark to represent for embedded application environments. The ARM binaries (i.e., source binaries) of the benchmarks were all compiled with GNU GCC 4.4.6 using the -O2 optimization setting. Besides, in order to reduce the size of the statically linked binary and translation time, we linked the benchmarks with μClibc [24] with the version 0.9.32. The LLVM version used in our HBT system is 3.0 which is the newest release version so far. The target binaries translated by our static translator use the LLVM default optimization setting (i.e., -O2). In order to reduce the dynamic translation time and keep reasonable code quality, the native code translated by our dynamic translator uses -mem2reg, -early-cse, -reassociate, -gvn, -dse, -simplifycfg, and -instcombine optimization settings [25]. The settings for our dynamic translator are selected because we have tested that they are most effective in our dynamic translation system. Benchmarks run on Intel Xeon E5506 (2.13GHz) with the Debian GNU/Linux operating system and Linux 3.2.0 kernel.

B. Performance

Figure 12 shows the execution time ratio of "DBT / HBT" in two different DBT configurations. "No JIT Opt" stands for without LLVM JIT optimizations which means the JIT compiler will disable all backend optimizations, use local register allocation, and use a fast instruction selection algorithm to generate native code as fast as possible. On the other hand, "Default JIT Opt" stands for using default backend optimizations which means the JIT compiler will use default global register allocation and normal instruction selection algorithm to make the quality of generated code as good as possible.

From the figure we can see that it is not worth to do backend optimizations in dynamic translators because the number of execution times of native code generated by the JIT compiler is too short to cover the optimization overheads. In addition, we can see that the performance difference between the HBT and DBT systems is large. Our HBT system can run 4 to 20 times faster than a DBT system. This is because our HBT system can perform many aggressive optimizations at static time that DBT systems can not. Our experimental results show that our HBT system can run about 8 times faster than DBT systems on average in the EEMBC benchmarks.

VI. Conclusion

DBT and SBT have their own merits and disadvantages. This paper presents a way to combine the merits of SBT and DBT and leverages the LLVM infrastructure to create a retargetable hybrid binary translator. Our translator translates

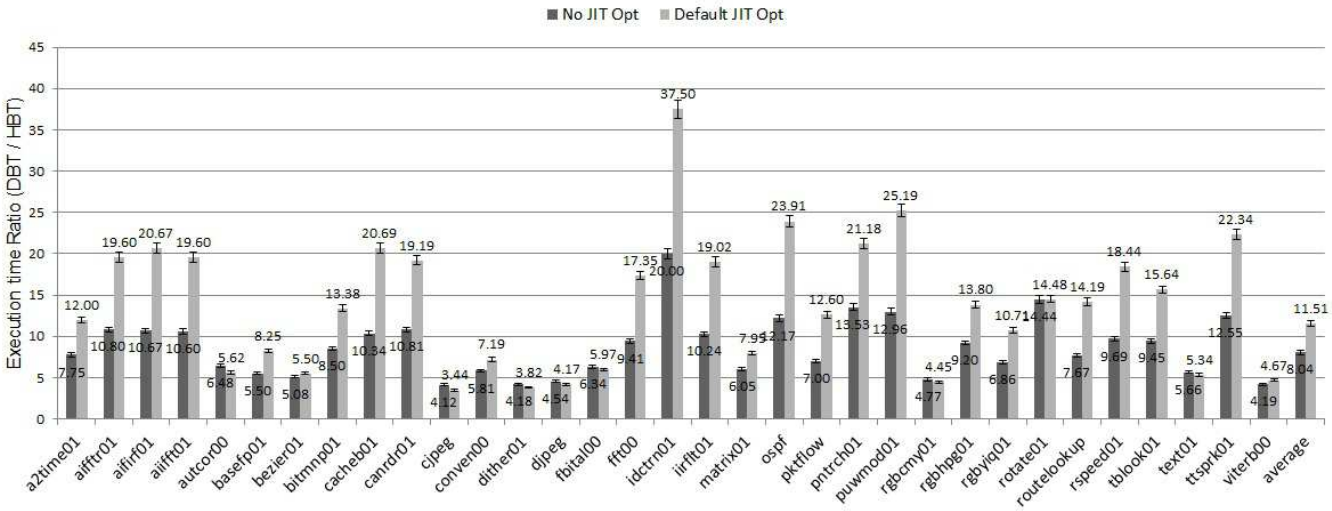

Fig. 12. Execution time ratio of "DBT / HBT" on translating EEMBC benchmark suites from ARM to X86.

and optimizes binaries statically so we can do time-consuming optimization passes at static time. In our experiment, we translated ARM binaries into X86 and compared our HBT system in hybrid with pure dynamic modes. The result shows that the HBT system can run about 8 times faster than DBT systems on average.

REFERENCES

[1] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, "Binary translation," *Commun. ACM*, vol. 36, pp. 69–81, February 1993.

[2] K. Andrews and D. Sand, "Migrating a CISC computer family onto RISC via object code translation," in *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-V. New York, NY, USA: ACM, 1992, pp. 213–222.

[3] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200.

[4] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

[5] Apple, "Rosetta," http://www.apple.com/rosetta/.

[6] C. Zheng and C. Thompson, "PA-RISC to IA-64: Transparent Execution, No Recompilation," *Computer*, vol. 33, pp. 47–52, March 2000.

[7] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates, "FX!32: A Profile-Directed Binary Translator," *IEEE Micro*, vol. 18, pp. 56–64, March 1998.

[8] C. Cifuentes and V. M. Malhotra, "Binary Translation: Static, Dynamic, Retargetable?" in *Proceedings of the 1996 International Conference on Software Maintenance*, ser. ICSM '96. Washington, DC, USA: IEEE Computer Society, 1996.

[9] E. Altman, D. Kaeli, and Y. Sheffer, "Welcome to the opportunities of binary translation," *Computer*, vol. 33, no. 3, pp. 40–45, March 2000.

[10] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–.

[11] C. Cifuentes and M. V. Emmerik, "UQBT: Adaptable Binary Translation at Low Cost," *Computer*, vol. 33, pp. 60–66, March 2000.

[12] C. Cifuentes, M. Van Emmerik, N. Ramsey, and B. Lewis, "Experience in the design, implementation and use of a retargetable static binary translation framework," Mountain View, CA, USA, Tech. Rep., 2002.

[13] J.-Y. Chen, W. Yang, C. Su, and W. C. Hsu, "A Static Binary Translator for Efficient Migration of ARM based Applications," in *Proceedings of the 6th Workshop on Optimizations for DSP and Embedded Systems*, April 2008.

[14] "LLVM-QEMU," http://code.google.com/p/llvm-qemu/.

[15] V. Chipounov and G. Candea, "Dynamically Translating x86 to LLVM using QEMU," Tech. Rep., 2010.

[16] A. Jeffery, "Using the LLVM Compiler Infrastructure for Optimised, Asynchronous Dynamic Translation in Qemu," University of Adelaide Honors Thesis, 2009.

[17] C.-C. Hsu, P. Liu, C.-M. Wang, J.-J. Wu, D.-Y. Hong, P.-C. Yew, and W.-C. Hsu, "LnQ: Building High Performance Dynamic Binary Translators with Existing Compiler Backends," in *Proceedings of the International Conference on Parallel Processing (ICPP'11)*, 2011.

[18] M. Gschwind, E. R. Altman, S. Sathaye, P. Ledak, and D. Appenzeller, "Dynamic and Transparent Binary Translation," *Computer*, vol. 33, pp. 54–59, March 2000.

[19] C. Cifuentes, B. Lewis, and D. Ung, "Walkabout: a retargetable dynamic binary translation framework," Mountain View, CA, USA, Tech. Rep., 2002.

[20] 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 *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003, pp. 191–.

[21] C. Lattner, "Intro to the LLVM MC Project," http://blog.llvm.org/2010/04/intro-to-llvm-mc-project.html.

[22] "GCC, the GNU Compiler Collection," http://gcc.gnu.org/.

[23] "EEMBC," http://www.eembc.org/.

[24] "μClibc," http://www.uclibc.org/.

[25] R. Spencer and G. Henriksen, "LLVM's Analysis and Transform Passes," http://llvm.org/docs/Passes.html.