Enhancing Cross-ISA DBT Through Automatically Learned Translation Rules

Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew
University of Minnesota, Twin Cities

Abstract

This paper presents a novel approach for dynamic binary translation (DBT) to automatically learn translation rules from guest and host binaries compiled from the same source code. The learned translation rules are then verified via binary symbolic execution and used in an existing DBT system, QEMU, to generate more efficient host binary code. Experimental results on SPEC CINT2006 show that the average time of learning a translation rule is less than two seconds. With the rules learned from a collection of benchmark programs excluding the targeted program itself, an average 1.25X performance speedup over QEMU can be achieved for SPEC CINT2006. Moreover, the translation overhead introduced by this rule-based approach is very small even for short-running workloads.

CCS Concepts • **Software and its engineering → Virtual machines**; **Dynamic compilers**; **Just-in-time compilers**; *Retargetable compilers*; *Translator writing systems and compiler generators*; *Runtime environments*;

Keywords DBT; Rule Learning; Symbolic Execution

ACM Reference Format:
Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. 2018. Enhancing Cross-ISA DBT Through Automatically Learned Translation Rules. In *ASPLOS '18: 2018 Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3173162.3177160

1 Introduction

Dynamic binary translation (DBT) is one of the core technologies in system virtualization. DBT has also been extensively used in many other applications such as whole program analysis and debugging, e.g., Pin [22], DynamoRIO [5], and Valgrind [25]; process/system virtualization, e.g., IA-32 EL [2] and QEMU [3]; vulnerability detection and defense, e.g., BitBlaze [30] and BAP [6]; mobile computation offloading [34] and computer architecture simulation, e.g., ZSim [27] and ARCSIM [20].

In general, a DBT system dynamically translates binary code from a *guest* instruction set architecture (ISA) to another *host* ISA, and executes the translated host binaries to emulate guest binaries on host machines. Previous work has shown that, in most cases, more than 90% of the execution time of a DBT system is spent on executing the translated host binary code [26, 33], i.e., the translation overhead is amortized by the frequent execution of the translated host code. Thus, the efficiency of the translated host code is crucial to the performance of the DBT system.

The binary translation process is typically driven by manually developed translation rules that map guest instructions into a sequence of semantically equivalent host instructions that can emulate the guest instructions on host machines [29]. Given the huge size of the modern ISAs, e.g., more than 1000 instructions in the x86 ISA [18], manual construction of translation rules is a major engineering challenge.

To ease the development and debugging efforts, most translation rules in existing DBT systems are constructed by mapping one guest instruction to one or several host instruction(s), i.e., one-to-one or one-to-many mappings. In this way, the total number of the translation rules required to translate the guest ISA can be limited. However, this also results in a sizeable code expansion in the translated host code because typically more than one host instructions are required to emulate one guest instruction due to the semantic differences between the guest and host ISAs. Moreover, intermediate representations (IRs) are often needed to ease the engineering efforts in code optimization and code generation, in particular in retargetable DBT systems that translate multiple guest ISAs to multiple host ISAs, such as QEMU. This process usually translates a guest instruction into several pseudo-instructions in IR, and each pseudo-instruction is then translated into one or more host instructions, causing further expansion in the final host binary code.

Taking advantage of architecture-specific instructions and generating efficient host binary code require substantial efforts in existing DBT systems, especially when the semantics of the host ISA are significantly different from those of the guest ISA. For instance, consider translating the following two guest instructions from ARM [28], which has a RISC architecture and is widely used in mobile devices, to an Intel

ASPLOS '18, March 24–28, 2018, Williamsburg, VA, USA

ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3177160

x86 host, which has a CISC architecture and is widely used in desktops and servers:

```
add r1, r1, r0
sub r1, r1, #1
```

QEMU, a popular and widely-used cross-ISA DBT system, needs to generate at least the following two x86 instructions to emulate the guest code sequence:

```
addl %eax, %edx
subl $0x1, %edx
```

Here, we assume the guest registers `r0` and `r1` are mapped to the host registers `eax` and `edx`, respectively. In fact, with a better understanding of the semantics of the ARM and x86 ISAs, we could find that those two ARM instructions are actually semantically equivalent to one x86 instruction:

```
leal -1(%edx, %eax), %edx
```

Although this kind of many-to-one or many-to-many translation rules can potentially produce more efficient host binary code, manual construction of such rules poses a significant engineering challenge due to their combinatorial increase and the required deep understanding of the semantics of both guest and host ISAs.

To improve the efficiency of the translated host binary for cross-ISA DBT systems, this paper presents a novel approach to intelligently learn translation rules from sample programs, where we compile the same source programs to produce both guest and host binaries. The learning process is fully automated with little manual intervention. The same source program is leveraged to bridge the semantic gap between the guest and the host ISAs.

More specifically, the source program is compiled twice (e.g., using LLVM [21]) to generate both the guest and the host binaries separately with the debugging flags turned on to provide the linkage between the source statements and their corresponding binaries. The guest and the host binary code snippets that come from the same line of the source code are then extracted automatically using the debugging information in the compiled binaries. They are used as the basis to learn the translation rules. The operands in the guest and the host binary code snippets are then parameterized to establish their mappings (see Figure 1 for an example, and the details are in Section 2). After that, the semantic equivalence of the guest and the host binary code snippets is verified formally using binary symbolic execution.

A prototype system using the learned translation rules has been built on QEMU to evaluate the effectiveness and the efficiency of this approach. Preliminary results show that the average time to learn a translation rule is less than two seconds. The prototype uses the learned rules to directly translate the guest binary to the host binary without going through QEMU's usual IR (called TCG ops), thus reducing the binary code expansion mentioned earlier. Using the rules learned from all benchmarks other than the target benchmark itself (i.e., the benchmark that is to be translated), we achieve 1.25X performance speedup on average for SPEC CINT2006 over QEMU. Moreover, it requires very little translation overhead to apply the learned rules even relative to short-running workloads, because the size of the rules is usually quite small (for larger rule sets, hierarchical or otherwise optimized lookup algorithms would help further, though our prototype does not use them). In particular, looking up the rules to generate a host binary code sequence from the corresponding guest binary sequence is much faster than a general translation that goes through an IR.

In summary, this paper makes the following contributions:

- We propose a novel approach to learn binary translation rules from available source programs using compilers' debugging facilities. It also leverages symbolic execution techniques to validate the semantic equivalence of the learned rules.
- We implement a prototype based on QEMU to demonstrate the effectiveness and efficiency of such a learning approach, and the performance of the host binary code generated by the learned rules. Moreover, several critical implementation issues are addressed.
- We conduct a number of comprehensive experiments. They show that the time to learn a translation rule is generally less than two seconds, and a 1.25X performance speedup can be achieved for SPEC CINT2006 on average over QEMU.

2 Issues and Challenges

Intuitively, the binary code compiled from the same program source code by the same compiler for different ISAs should be equivalent in program semantics. This observation inspires the learning approach proposed in this paper, which leverages the same program source code to bridge the semantic gap between the guest and the host ISAs. Although the idea behind the learning approach is rather simple, there are still several significant technical issues and challenges to put it into practice.

Learning Scope. We need to select an appropriate learning scope that determines the range of guest/host code snippets (i.e., sequences of guest and host binary instructions) that will be involved in the rule extraction. Since we use the source program to bridge the semantic gap between the guest and the host binary codes, the learning scope is determined by the statements at the source code level.

Considering the basic translation unit of most DBT systems is a *basic block* (or *block* for short), which is a straight-line guest binary code sequence with only one entry and one exit, a possible choice of the learning scope is a basic block in the source program. However, it is very common for compilers to apply various transformations/optimizations across multiple source code blocks, e.g., loop transformations, which makes it very hard to map binary code blocks directly back to their common source code blocks. Moreover, the same source code block could generate multiple binary

code blocks due to different transformations/optimizations applied in different ISAs (even by the same compiler). In addition, a large learning scope may involve a large number of guest instructions in the learned rules, making a rule less likely to be applied in practice because it is rare to exactly match a long sequence of guest binary code.

Based on the above considerations, we propose to use a *line of source code* (*source line* for short) to determine our learning scope. To extract the groups of guest and host instructions that correspond to the same source line, our approach leverages the *debugging information* provided by almost all existing compilers, which is originally produced to enable source-level debugging. It provides information about which source line a binary instruction is generated from. In our implementation, LLVM is selected because the debugging information is maintained quite comprehensively [8] even after many transformation passes are applied. As we will show later, our approach is insensitive to the compiler selected for learning, i.e., the rules learned using one compiler can be applied to guest binaries generated by another compiler.

Parameterization of Operands. After the guest and the host instruction sequences that correspond to the same source line are identified, the next task is to check whether they produce semantically equivalent machine states from equivalent initial machine states. The challenge here is to find out whether a feasible mapping exists between the operands in the corresponding guest and host instructions such that the guest operands can be emulated by the corresponding host operands under this mapping. Here, we assume a host register can hold no fewer bits than those of a guest register, and the guest and host architectures have the same endianness. Additional work might be required if these assumptions are not satisfied, but it is beyond the scope of this paper.

To overcome this challenge, our approach heuristically generates an ***initial mapping*** between the guest and the host memory operands, live-in registers, and immediate operands. We then verify the semantic equivalence of the guest and host binary code sequences under this mapping. This process could be iterated multiple times with different initial mappings.

To generate an initial mapping, our approach takes advantage of the information available for the memory operands in the IRs generated by the compilers, e.g., LLVM IRs, to establish mappings between guest and host memory operands. With the aid of the mapped memory operands, the live-in registers used to calculate a memory address, including its related immediate values, e.g., offset, are mapped by normalizing the calculation of the memory address (Section 3.2). In this way, our approach can map up to 80% of live-in registers and immediate operands. For the remaining live-in registers and immediate operands, we heuristically generate a mapping based on the operations performed on them or based on their values (Section 3.2).

It is not surprising that there could be no feasible mapping for some guest and host instruction sequences because, for example, they may have different numbers of memory variables or live-in registers. Those sequences are simply discarded as they are considered as inappropriate candidates for translation rules. This only affects the *yield* of the harvested rules, but makes the time to learn rules much faster. It has minimum impact on the *coverage* and the *quality* of the harvested rules as shown in Section 6.2.

Verification of Semantic Equivalence. Even given a probable initial mapping of operands, it is still quite challenging to know whether the guest and the host instruction sequences are semantically equivalent or not. In theory, we could test them with the same but randomly selected input machine states on real machines. However, technically, it is extremely hard to use such an approach to ensure a *complete* coverage in the testing. Instead, our approach is to verify the equivalence by *symbolically executing* the guest and the host instructions under the same initial machine states.

Symbolic execution [4] is a popular program analysis technique in software testing to determine what inputs can cause each part of a program to execute. It executes a program with *symbolic* rather than concrete input values and thus can simultaneously explore multiple paths that a program could take under different input values. This enables symbolic execution to achieve a complete coverage in verification.

In our approach, the equivalence is verified by evaluating the symbolic execution results of *registers*, *memory operands*, and *branch conditions*. First, the symbolic values of all guest and host registers that are updated in their sequences are evaluated to find a ***final mapping*** between them such that the updated guest registers can be emulated by the corresponding updated host registers under this mapping, and the final mapping should not conflict with the initial mapping. Second, as the memory operands are mapped in the initial mapping, the symbolic values stored in the mapped memories are evaluated to see if they are equivalent. Last, if the last guest and host instructions in their sequences are conditional branches, the symbolic values of the conditions that trigger the branches are evaluated to check their equivalence. If the verification passes all three aspects described above, a translation rule that maps the guest instruction sequence to the host instruction sequence is established ("learned"). Otherwise, the same verification process is iterated if a different initial mapping can be generated heuristically. More details are presented in Section 3.3.

In Figure 1, we use an example to show such a rule-learning process, in which the source code in line *12748* of *util.c* is compiled into two guest (ARM) instructions and one host (x86) instruction by LLVM-ARM and LLVM-x86 compilers, respectively. The corresponding LLVM IRs generated during the compilation are also shown (note: simplified for presentation). Note that we use the *machine specific* IRs instead of the *high-level* IRs in the compilers, which allows us to

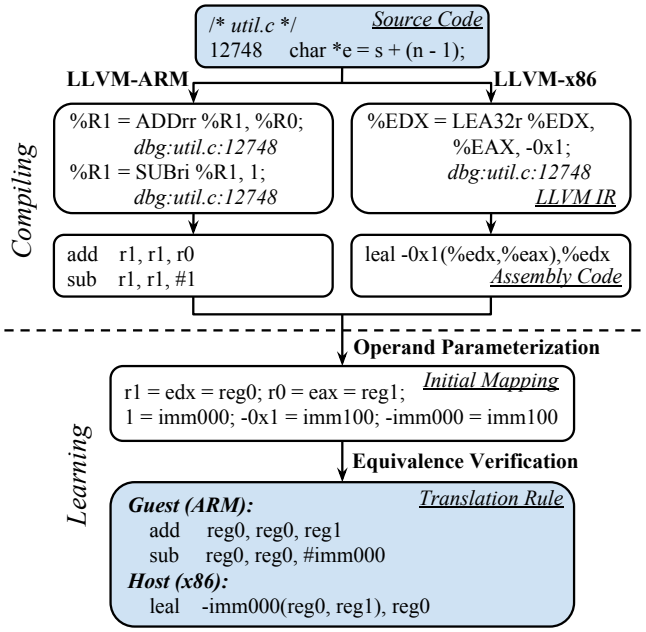


Figure 1. The learning process of translation rules.

establish an almost one-to-one mapping between the binary instructions and their corresponding IRs [14].

Since there is no memory access in this example, we heuristically generate an initial mapping. A probable mapping for the live-in registers is: `r1` ↦ `edx` and `r0` ↦ `eax`, or vice versa, as the operation on the live-in registers is an *addition* (which is commutative). The immediate operands can be mapped using an *additive inverse* operation according to the values, i.e., *-imm000* ↦ *imm100*. Then, the guest and the host instructions are symbolically executed under this mapping. Obviously, the parameterized guest and host registers, i.e., `reg0` and `reg1` for `r1` and `edx`, have the same symbolic results under this initial mapping. Also, there is no memory operand and branch condition required to be verified. Thus, a parameterized translation rule that translates the guest (ARM) instruction sequence with two instructions, "*add reg0, reg0, reg1*; *sub reg0, reg0, #imm000*", to one host (x86) instruction, "*leal -imm000(reg0, reg1), reg0*", is learned.

3 Learning Binary Translation Rules

There are four key requirements on learning binary translation rules from program source code.

- **Automatability:** The learning process should be automated, and require as little human supervision and manual manipulation as possible.
- **Semantic Equivalence:** The semantics of the guest and the host instruction sequences in the learned translation rules should be equivalent.
- **Parameterizability:** The register/immediate operands that appear in the learned translation rules should be parameterized as much as possible to enhance the coverage of the rules.
- **DBT Independence:** The learning process should be independent of the DBT system that uses the learned rules. This allows the learned rules to be applied to most DBT systems.

The learning approach proposed in this paper satisfies all above requirements. It automatically learns translation rules from guest and host instruction sequences that correspond to the same source line. Details about the learning process are presented in the following subsections.

3.1 Preparation for Learning

Commonly, a line of source code contains one statement in most high level languages. But some source lines may include multiple statements, e.g., because of macros in C. Also, source code from different applications may have a wide variety of programming styles. Hence, it is necessary to utilize the compiler to preprocess the source code, and put it in a more regular format that is easier to trace back from the binaries. In our approach, we use the "-E" option of LLVM to preprocess the source code and the *clang-format* tool provided by LLVM to format the source code.

We also exclude some special cases in guest and host instruction sequences. For example, we ignore guest and host instruction sequences that contain *call* or *indirect branch* instructions because these instructions need to be handled specifically in most DBT systems. Also, due to the time and the resource constraints, our prototype currently does not support (ARM) predicated instructions and guest/host instruction sequences that are composed of multiple blocks. However, we believe there should be no particular technical difficulties to include such features in the future.

3.2 Parameterization of Operands

The key to parameterize operands is to establish a mapping between guest and host operands. To keep the learning process simple and efficient, our approach heuristically generates *initial mappings* based on the following principles.

First, we only map guest and host operands of the same types. There are typically three different types of operands in most modern ISAs: *register*, *memory*, and *immediate value*. It could potentially complicate the learning process if mappings across different operand types are allowed because the opcodes may have to be changed in some ISAs if the types of the operands are different.

Second, arithmetic/logical operations are introduced to accommodate the differences in values between the guest and host immediate operands. Recall the example in Figure 1, in which the immediate operands *imm000* and *imm100* are mapped using an *additive inverse* operation.

Last, each guest/host register should have only one mapped host/guest register in the initial mapping. The reasons of

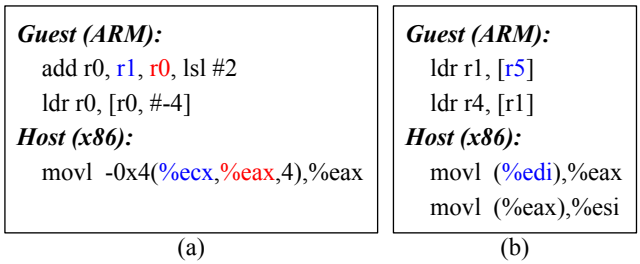


Figure 2. Live-in register mapping using memory operands.

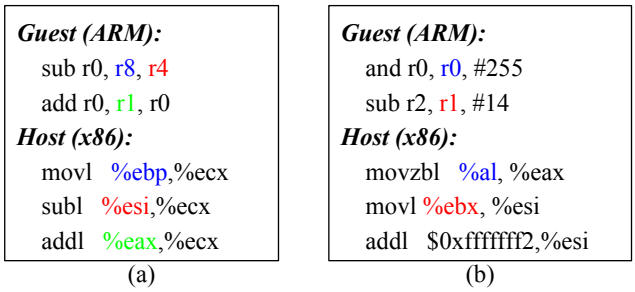


Figure 3. Live-in register mapping using operations.

this principle are obvious, i.e., one guest register should be emulated by only one host register, and one host register cannot emulate multiple guest registers at the same time.

Based on these principles, we next describe how to generate an initial mapping for each type of operands.

Memory Operands. The guest and host memory operands are mapped according to the names of the corresponding variables in LLVM IRs. This is reasonable because the guest and the host binary codes are compiled from the same source code, and generally, most variable names in LLVM IRs are inherited from the source code.

Live-in Registers. A guest/host register is considered to be *live-in* if it is *used* before being *defined* in the guest/host instruction sequence. All other non-live-in registers are *defined* in the instruction sequences before their *use*. Note that a live-in register can be defined again in the instruction sequence after its use. We only map live-in registers in the initial mapping. The defined registers will be mapped after the symbolic execution (see Section 3.3).

For live-in registers used in memory addressing modes, which account for up to 80% of all live-in registers, the mapping can be established after being normalized. There could be many addressing modes supported by modern ISAs. However, most addressing modes can be normalized (or standardized) to the following formula:

$$base \pm index \times scale + offset$$

where *base* and *index* can be live-in registers or other memory operands, and *scale* and *offset* are immediate values.

Figure 2 shows two examples of mapping live-in registers using normalized memory addresses. In Figure 2(a), the guest memory operand `[r0,#-4]` and the host memory operand `-0x4(%ecx,%eax,4)` correspond to the same variable in the LLVM IRs. Since `eax` and `ecx` are live-in registers in the host instruction sequence, it is quite easy to normalize the host addressing mode as:

$$\texttt{ecx} + \texttt{eax} \times 4 + (\texttt{-0x4})$$

However, even though `r0` is a live-in register in the guest instruction sequence, it is defined by the first instruction in the sequence. In this case, we can find the live-in registers (or memories) that redefine `r0` from LLVM IRs. In this example, we can find that `r0` is defined as: $\texttt{r1} + \texttt{r0} \times (1 \ll 2)$, where `r1` and `r0` are live-in registers with initial live-in values. So, the guest memory addressing mode can be normalized to:

$$\texttt{r1} + \texttt{r0} \times (1 \ll 2) + (\texttt{-4})$$

Here, we keep $(1 \ll 2)$ instead of using the value 4 in order to map immediate operands later. After normalizing the two memory addresses, the mapping of the live-in registers can be found as follows: $\texttt{r1} \mapsto \texttt{ecx}$ and $\texttt{r0} \mapsto \texttt{eax}$.

Similarly, in Figure 2(b), we have $\texttt{r5} \mapsto \texttt{edi}$. Note that, in this example, no mapping is established for `[r1]` and `(%eax)` even though they are from the same variable in LLVM IRs. It is because `r1` and `eax` are not live-in registers, i.e., they are defined before their use. Such mappings will be established after symbolic execution, as mentioned earlier.

Live-in registers that are not in the address calculation can be heuristically mapped based on the operations performed on them. Typically, the common operations include *addition*, *subtraction*, *multiplication*, *division*, *logical not/and/or*, and *logical left/right shifting*. Figure 3(a) shows such an example. Here, the operation performed on `r1` is an addition, which is the same as the operation performed on `eax`. Therefore, we can heuristically assign its initial mapping as: $\texttt{r1} \mapsto \texttt{eax}$. Similarly, the following two mappings can be found: $\texttt{r8} \mapsto \texttt{ebp}$ and $\texttt{r4} \mapsto \texttt{esi}$ because they are in the corresponding subtraction instructions.

Unfortunately, such a heuristic approach does not always work because of the diverse operations available in the guest and the host ISAs. In Figure 3(b), the operation on `r0` is a *logical and*, while there is no corresponding logical operation on `al` (i.e., `eax`) because the compiler selected the instruction `movzbl` for the host machine. Furthermore, the operation on `r1` is a subtraction while the operation on `ebx` is an addition. Actually, in this case, it is not hard to check manually that the guest and the host instruction sequences are equivalent. However, to do it automatically without human intervention, if the guest and the host instruction sequences have the same numbers of live-in registers, and in most cases this number is quite small, we can exhaustively try all possible permutations between two sets of live-in registers for such mappings, and select a plausible one. We then rely on the symbolic execution step to validate the selected mapping. If the number of live-in registers are different, we discard them without proceeding further, i.e., no rule will be generated.

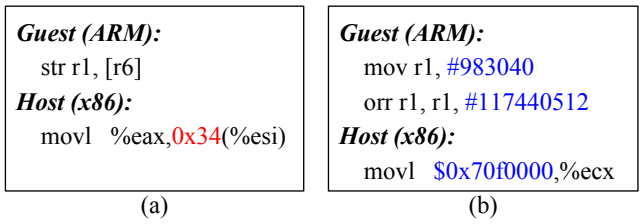


Figure 4. Immediate operand mapping with operations.

In Figure 3(b), only two mappings are possible: `r0` ↦ `ebx`, `r1` ↦ `eax` and `r0` ↦ `eax`, `r1` ↦ `ebx`. In our implementation, we limit it to 5 tries, which is sufficient in most cases.

Immediate Operands. Immediate operands are mapped in a similar way to live-in registers. The only difference is that arithmetic/logical operations can be used to accommodate the differences between the guest and host values.

The immediate operands in the normalized addresses are mapped as follows: $scale_{guest} \mapsto scale_{host}$ and $offset_{guest} \mapsto offset_{host}$. Recall the example in Figure 2(a), where the guest has a scale of $(1 \ll 2)$ and the host's is 4, so they can be mapped as $(1 \ll 2) \mapsto 4$. Also, the offsets can be mapped as `-4` ↦ `-0x4`. Here, the values of the immediate operands for both the guest and the host are the same. But, this may not be always true. As shown in Figure 4(a), the guest has an offset of `0`, while the host has an offset of `0x34`. Thus, we have the following mapping: `0` ↦ `0x34`, which means the host offset is parameterized to `0` in the initial mapping.

The immediate operands that are not used in memory address calculations are mapped based on their actual values. It is quite common that the guest values are different from the host values. To tackle this problem, arithmetic/logical operations can be introduced that include *addition*, *subtraction*, *multiplication*, *division*, *additive inverse*, *logical not/and/or*, and *logical left/right shifting*. Figure 4(b) shows an example in which the value of either of the two guest immediate operands is different from the value of the host immediate operand. After trying different operations, we can derive that the host value can be calculated by the two guest values using a *logical or* operation. Thus, they can be mapped as (`983040` | `117440512`) ↦ `0x70f0000`. It is worth pointing out that the same mapping could be concluded even if x86 is the guest ISA and ARM is the host ISA in this example. However, additional work might be required when the host ARM instructions are assembled because of the limited value ranges that can be encoded for the immediate operands. More details about this work are discussed in Section 5.

It is possible that there will still be unmapped guest/host immediate operands after the above steps. There are two such scenarios. In the first scenario, the immediate operands should not be parameterized in the first place. Recall the example in Figure 3(b), where the guest immediate operand `14` and the host immediate operand `0xfffffff2` can be mapped using an *additive inverse* operation: `-14` ↦ `0xfffffff2`, while the guest immediate operand `255` is left unmapped. In fact, the `movzbl` instruction implements the same semantics as the `and` instruction in a different way. Therefore, `255` should not be parameterized. In the second scenario, no operation can make the guest and host values equal. Hence, the immediate operands are left without being parameterized. In our experience, these cases are very rare.

All of the rule candidates obtained after the parameterization and initial mapping need to be verified in the symbolic execution phase described in the next subsection.

3.3 Verification of Semantic Equivalence

To check the parameterization and the initial mapping generated in the previous subsection, as well as to confirm the equivalence of the instruction sequences and to establish a final mapping, we next verify a rule candidate's correctness using binary symbolic execution.

Two instruction sequences are equivalent if they produce equivalent outputs whenever their inputs are identical, so we represent the corresponding input machine state using matching *symbols*. Our system then symbolically executes both instruction sequences and checks, using an SMT solver [12] if necessary, whether the corresponding output *symbolic expressions* are equivalent or not. Specifically, the equivalence is checked from three aspects:

Registers. The symbolic values of all defined guest and host registers are evaluated to establish a *final mapping* such that each defined guest register is equivalent to a different defined host register. Then, the final mapping is compared with the initial mapping to see if there is any conflict. A conflict occurs if a guest/host register is mapped to different host/guest registers in the initial and the final mappings, which implies the initial and the final mappings are inconsistent. Such a conflict might be incompatible with the register allocation scheme in the DBT system that uses the learned rules, so rule candidates with conflicts are discarded.

Memory Operands. Since the guest and the host memory operands have been mapped under the initial mapping, we only need to evaluate the symbolic values stored in the corresponding memory locations to see if they are equivalent. One subtlety here is that the registers used in address calculations may also be modified, changing the meaning of an address expression using that register. To avoid this problem, we record the symbolic expressions corresponding to the memory access addresses when they are used, and use these recorded locations for the memory equivalence check.

Branch conditions. If the last guest and host instructions in their sequences are conditional branches, the symbolic values of the conditions that can trigger the branches are checked for equivalence. Here, we assume that the guest and the host branch targets are the same. Branch conditions are typically combinations of condition code flags, but here we only check whether the final conditions are equivalent or

not. For instance, there is no check on condition codes that are not used in a branch. For the correctness of condition codes in other situations, we use a mechanism described in Section 5.

If the verification passes all above mentioned three criteria, a translation rule that maps the guest code sequence to the host code sequence is learned. Otherwise, the same process will be iterated if a different initial mapping can be generated.

4 Applying Translation Rules

To apply the learned translation rules to a DBT system, the rules are installed in a hash table at the start of the system. We simply use the arithmetic mean of the guest opcodes in each rule as its hash key. More sophisticated hash schemes will be explored in our future work. When a guest code block is translated, the hash table is searched to find the rules that match the guest instructions in this block.

A contiguous guest instruction sequence in the block is examined to match with the rules that have the same hash key. The maximum length of the sequence starting from the ith instruction in the block is $n - i$, where n is the number of guest instructions in the block and $0 \leqslant i < n$. We start from the first guest instruction and examine if n guest instructions starting from it can match any rule. If not, a shorter length will be tried, until the length is 1. If there is no rule that can match a contiguous guest instruction sequence starting from the current one, we move to the next until all guest instructions are examined or a matching rule is found.

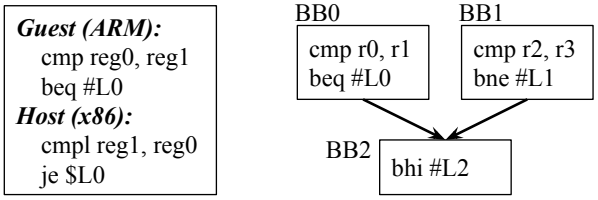
After a matching rule is found, the parameterized operands in the rule are bound with the corresponding operands in the matched guest instructions. Then the host instructions in the matched rules are assembled to executable binary code. Guest instructions that have no matching rule will be translated through the regular translation process.

5 Implementation

The learning process described above has been implemented in a prototype that comprises around 3.6K lines of Python code. It takes two significantly different ISAs, i.e., ARM and x86, as the guest and the host ISAs. It is worth noting that ARM and x86 are prominent representatives of RISC and CISC ISAs, respectively. The guest binaries are built by LLVM version 3.8.0 via cross compilation on the host machine.

The verification is implemented based on the binary symbolic execution engine FuzzBALL [24], which interprets each binary instruction by translating it into the BitBlaze [30] Vine IR. The equivalence of symbolic expressions is checked by the SMT solver STP [12]. To invoke FuzzBALL, we have implemented a wrapper, which includes around 1.2K lines of OCaml code, to parse the initial mapping, initialize the input machine states, and extract the execution results.

Next, we describe how to extend an existing DBT system to utilize the learned translation rules. We use QEMU



(a) A translation rule (b) Control flow of guest code

Figure 5. Condition code emulation of the translation rules.

(version 2.6.0, which is the latest version at the start of the implementation) to demonstrate our approach, because it is open source and has been widely used in academia and industry. It is worth noting that the learned translation rules are independent of QEMU, and can be used by most existing cross-ISA DBT systems with similar extensions.

A tiny code generator (TCG) is used by QEMU to translate guest binaries into host binaries with TCG OPs as the IR. Since TCG is still needed to translate guest instructions that have no matching rules, it is vitally important for our implementation to work with TCG seamlessly.

Register Allocation. In TCG, the ARM guest registers are emulated using host memory locations, and the consistency is maintained at the boundaries of blocks. For each accessed guest register in a block, a host register is allocated and associated with the corresponding host memory location. A liveness analysis determines when to write back the host register to the corresponding host memory location.

To simplify the implementation, we reuse the register allocator in TCG. For guest instructions to be translated using the rules, we gather the written-back information for each accessed guest register. When the host instructions obtained from the rules are assembled, the register allocator in TCG is invoked to allocate host registers for the accessed guest registers. Additional host instructions may be generated to load from the host memory locations corresponding to the accessed guest registers if they have not been loaded yet. Also, the written-back information is examined to see if it is necessary to generate additional host instructions to write back the allocated host registers to the corresponding memory locations.

Condition Codes. Both ARM and x86 have condition codes (known as EFLAGS in x86), which are extra bits kept by a processor to summarize the results of an instruction, and to affect the execution of later instructions. How to support the rules associated with condition codes is another key issue in our implementation.

TCG emulates each of the four ARM condition codes, i.e., negative (*NF*), zero (*ZF*), carry (*CF*), and overflow (*VF*), in a way similar to the general-purpose registers, i.e., using host memory locations to maintain the consistency across blocks. Besides such schemes, translation rules can also directly emulate guest condition codes using host condition codes

with equivalent semantics. Figure 5(a) shows a rule with condition codes, where the guest instructions `cmp` and `beq` are emulated by the host instructions `cmpl` and `je` as the guest *ZF* can be emulated by the host *ZF*. Since the learned rules may not cover all guest instructions with condition codes, our implementation needs to cooperate with TCG to achieve the correct emulation of guest condition codes.

Figure 5(b) shows an example of using rules to translate guest instructions with condition codes. Suppose the block `BB0` is translated by the rule in Figure 5(a) while `BB1` is translated by TCG. Then, how to translate `BB2` is a key challenge as `bhi` in `BB2` can use condition codes defined in either `BB0` or `BB1`. A possible solution is to generate additional host instructions at `BB0`, as TCG does, to maintain host memory locations used to emulate guest condition codes. However, this will introduce significant performance overhead. Instead, we generate additional host instructions to save *host condition codes* at `BB0`, which requires only 3 host instructions. For `BB2`, our implementation generates *two versions* of host binary code that correspond to the rule-based version and the TCG translated version, respectively. A boolean flag is then added to determine which version to execute at runtime.

Another potential issue is that the condition codes generated by the guest and the host instructions in a rule can be inconsistent. For instance, consider a rule with `adds reg0, reg0, #1` as the guest instruction (ARM) and `incl reg0` as the host instruction (x86). The guest condition code *CF* cannot be emulated by the host condition code *CF*, as `incl` does not update the host *CF*. To tackle this problem, the verification tool is extended to output guest condition codes that cannot be emulated by host condition codes. In our implementation, a lightweight analysis is performed at translation time to see whether the guest condition codes that are not emulated are used by the following guest instructions or not. If not, the translation rules can be applied normally.

Host ISA Specific Constraints. Typically, there are some ISA specific constraints such as the kinds of operands that can be used for certain opcodes, the kinds of registers/values that can be used as operands, or the value ranges that can be encoded by the immediate operands. For example, the memory operands in the x86 ISA can only have 1, 2, 4, or 8 as the scale values. To satisfy such specific host ISA constrains, our implementation examines the host instructions for such rules before assembling them, and generates additional host instructions if necessary.

6 Experimental Results

Our learning prototype is evaluated using SPEC CINT2006, which is an industry standard benchmark suite. In the SPEC CINT2006, it includes a large number of benchmark programs from a wide range of different application domains that include artificial intelligence, search engine, quantum computing, video compression, XML processing, C compilation, etc. To be as close to the real-world usage scenarios as possible, the performance of each benchmark program is evaluated using the translation rules learned from all other benchmark programs that do not include the evaluated benchmark program itself. To demonstrate that our learning approach is insensitive to the compiler selected for learning, we also apply the learned translation rules to the guest binaries built by GCC (version 4.7.3) [13] as our rules are learned from LLVM generated binaries as described earlier.

To demonstrate the efficiency of the host binary code generated by the learned translation rules, we also implemented another DBT system based on QEMU. It employs LLVM JIT as the backend to generate optimized host binary code. To minimize the engineering effort, we firstly translate TCG Ops into LLVM IRs, and then invoke LLVM JIT to generate optimized host binary code. With the optimizations available in LLVM JIT, it is expected that more efficient host code can be produced. For a fair comparison, both the learning system and the LLVM JIT in this implementation use the same optimization level of LLVM, i.e., "-O2". The impact of different compiler optimization levels on the learning results will be discussed later.

The experimental platform is equipped with an Intel Core i7-4870HQ processor and 16GB memory. The operating system is 32-bit Ubuntu 14.04 with Linux 3.13. In experiments related to performance, the platform is set up exclusively to run the evaluated benchmarks. Moreover, each benchmark is run three times, and their arithmetic mean is used to reduce the influence of random factors.

6.1 Learning Results

Table 1 shows the results of our learning approach. The number of guest and host instruction sequences that cannot pass a particular learning step, which includes *preparation*, *parameterization* and *verification*, is listed under its corresponding label with separate columns according to the cause of the failures. For instance, the column "CI" under "#F in Preparation" shows the number of sequences failed in the learning preparation step due to the call or indirect branch instructions in the sequences. As shown in the table, 43% of the code sequences failed in the preparation step, i.e., there are around 57% left for the remaining steps to learn translation rules. As mentioned before, our learning prototype currently does not support (ARM) predicated instructions and guest/host instruction sequences composed of multiple blocks due to the time and resource constraints. So there is still ample room for improvement in this step, which is left as part of our future work.

The columns under "#F in Parameterization" show the numbers of instruction sequences that failed to produce a probable initial mapping. They include the instruction sequences that have different numbers/names of memory variables (shown in columns "Num" and "Name", respectively) in

	PL	LoC	#F in Preparation			#F in Parameterization			#F in Verification				#Rules	Time
		(K)	CI	PI	MB	Num	Name	FailG	Rg	Mm	Br	Other		(S)
perlbench	C	128	13887	2964	287	3864	639	4252	6549	603	1947	471	**13171**	18206
bzip2	C	5.7	361	243	41	156	21	590	431	52	53	44	**818**	1249
gcc	C	386	44978	9966	485	10812	1404	11670	11492	896	4003	966	**36325**	34976
mcf	C	1.6	93	44	1	44	8	61	53	8	7	6	**185**	141
gobmk	C	158	8459	1272	68	5030	480	3451	2206	228	374	119	**3385**	4133
hmmer	C	40.7	3898	551	54	585	188	1094	742	151	193	50	**1808**	2299
sjeng	C	10.5	1057	425	33	469	111	766	665	101	83	49	**735**	1362
libquantum	C	2.6	532	61	3	80	12	53	51	5	5	6	**125**	122
h264ref	C	36	3824	1430	96	2061	244	3398	2880	353	341	399	**3284**	7440
omnetpp	C++	26.7	7450	350	25	289	44	423	589	37	97	55	**1470**	1792
astar	C++	4.3	456	119	8	108	26	193	146	13	20	14	**292**	315
xalancbmk	C++	267	34654	2752	248	3446	254	4612	5875	630	1709	699	**16161**	17916
					43%			19%				14%	**24%**	

Table 1. Learning results. Number of failures (#F) in preparation due to "CI": call or indirect branch instructions, "PI": predicated instructions, and "MB": multiple blocks; Number of failures in parameterization due to "Num/Name": different numbers/names of memory variables in guest and host LLVM IRs and "FailG": failed to generate initial mappings for live-in registers; Number of failures in verification due to "Rg/Mm/Br": inequivalent registers/memories/branch conditions.

LLVM IRs. Another reason is that our approach fails to generate initial mappings for live-in registers in guest and host instruction sequences, e.g., due to different numbers of guest and host live-in registers (shown in column "FailG"). Overall, only 19% of total sequences failed in this step, i.e., 67% of the remaining sequences after the preparation step can successfully produce a probable initial mapping for learning rules. This shows the effectiveness of the proposed heuristic approach to parameterize the operands.

Failures in the verification step usually result from inequivalent guest and host registers (column "Rg"), memory operands (column "Mm"), or branch conditions (column "Br"). Other failures that include timing out and system crashes during binary symbolic executions (due to the limitation of the binary symbolic execution engine) are listed in the "Other" column under the label "#F in Verification". One thing to note here is that for instruction sequences that have multiple verified initial mappings, only their last verification results are counted. From the table, we can see that the main source of semantic inequivalence is from registers. This is reasonable because the compiler may apply different transformations/optimizations for guest and host ISAs due to different numbers of available registers in the ISAs.

On the whole, our system can successfully learn binary translation rules from every evaluated benchmark. Through the process, the learned translation rules can be obtained from around 24% of the total guest and host instruction sequences. We call it the "yield" of our learning process. After investigating the learned rules, we found that the length of most learned rules, i.e., the number of guest instructions in the rules, is between 1 and 6, which shows the suitability of our learning scope since this length is typically smaller than most of blocks translated by a DBT system [29].

Another interesting observation is that many of the rules learned from the same benchmark have the same guest and host instruction sequences. This implies there are often many same/similar code structures in the same program source code. It also provides a strong evidence that our operand parameterization approach has worked effectively to generalize different operands, e.g., registers. However, there are some rules that have the same guest instruction sequence but with different equivalent host instruction sequences. They are redundant from the perspective of a DBT system because only one of them is needed to translate the matched guest binary. In this case, we select the sequence with the smallest number of host instructions based on the observation that the number of host instructions executed correlates strongly to the performance of the generated host code.

In addition, the last column in Table 1 shows the learning time (in seconds) needed for each benchmark. Obviously, it takes different amount of time to learn translation rules from different benchmarks due to their different program sizes. In general, less than one hour is required to learn translation rules from most of the benchmarks, e.g., *bzip2* and *mcf*. For some benchmarks with large code bases, e.g., *gcc* and *xalancbmk*, our system can still finish the learning process in several hours. On average, the time to learn one translation rule using our approach is *less than two seconds*, which demonstrates the efficiency of our learning approach.

By looking at the time needed in each step of our learning process, we found that around 95% of the time is spent in the verification step. Our current implementation run the entire SPEC CINT2006 benchmarks as a batch through each step of the learning process. Obviously, each benchmark program can be processed concurrently to substantially speedup the entire learning process if multi-core resources are available.

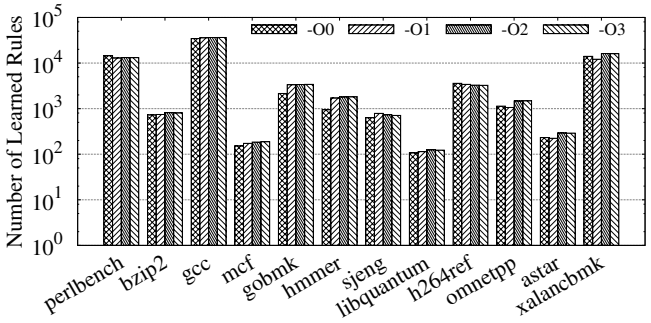


Figure 6. Sensitivity of learning on optimization levels.

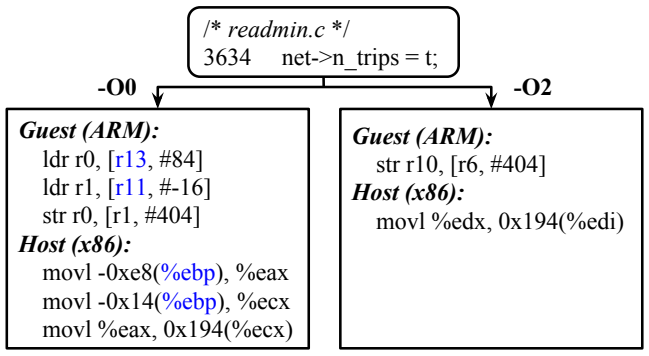


Figure 7. Different optimization levels for learning rules.

Figure 6 shows the number of translation rules learned using different optimization levels of LLVM to compile the program source code. Interestingly, for some benchmarks, e.g., *gobmk* and *hmmer*, more translation rules can be learned at higher optimization levels. After investigating the learning process, we found that higher optimization levels can produce more simplified (i.e., optimized) binary code, which improves the success rate of learning rules. Figure 7 shows such an example, where a translation rule can be learned from the source line using "-O2", but the guest and host live-in registers cannot be mapped using "-O0" due to different numbers. Overall, a similar number of translation rules can be learned with different optimization levels. This demonstrates that such a learning approach is not very sensitive to the optimization levels used to compile the source code.

6.2 Performance Study

Figure 8 shows the performance comparison between our prototype using the learned translation rules and the implementation using LLVM JIT as the backend, where the guest binaries are built by LLVM. The performance of the original QEMU is used as the baseline.

For the short-running *test* workload of SPEC CINT2006, it is hard to benefit from LLVM JIT because of the heavy translation overhead introduced by LLVM JIT. In contrast, our prototype introduces quite small translation overhead

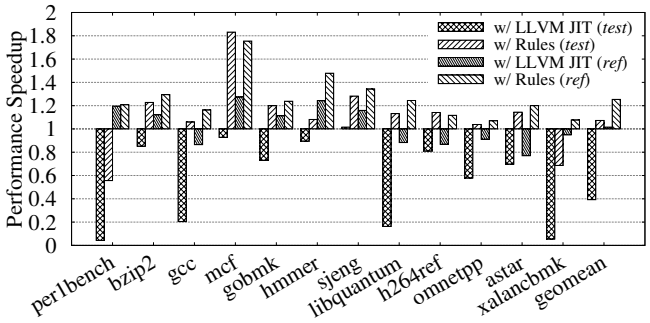


Figure 8. Performance speedup for guest LLVM binaries.

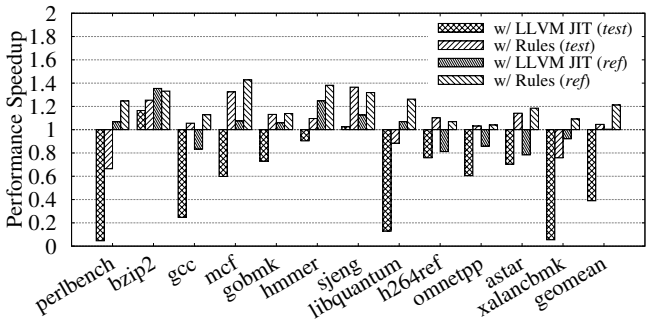


Figure 9. Performance speedup for guest GCC binaries.

by using the learned translation rules. It allows most of the benchmarks to benefit from the learned rules except *perlbench* and *xalancbmk*, whose execution time is less than 1 second. Overall, our prototype achieves 1.07X performance speedup on average for the *test* workload while the LLVM JIT suffers 0.39X slowdown due to translation overheads.

Typically, the translation overhead can be amortized in long-running workloads. In such cases, the *efficiency/quality* of the translated host code is the main factor that impacts the overall performance. As shown in Figure 8, LLVM JIT achieves 1.02X performance speedup on average with the *reference* workload, while our prototype attains a performance improvement for all benchmarks with an average of 1.25X speedup. This shows that the host binary code generated by the learned rules can be quite efficient and of high quality.

After studying the host binary code generated by both the learned rules and LLVM JIT, we have the following two observations. First, additional register spills could be introduced by the register allocator in LLVM JIT due to the use of host memory to maintain a copy of the guest register file. Second, lack of global information limits the ability of LLVM JIT to carry out more aggressive program optimizations. In contrast, our prototype is less impacted by such limitations, as it leverages the learned translation rules and cooperates with TCG to generate efficient host binary code.

To study the sensitivity of the compilers that are used to produce the translation rules when those rules are used to

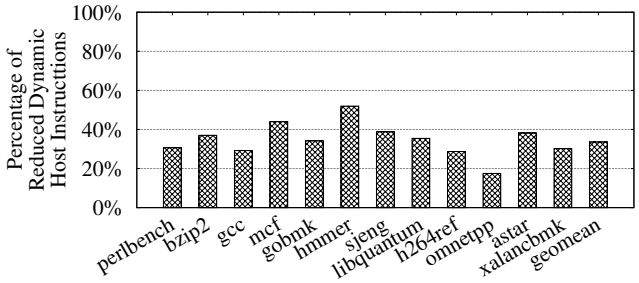


Figure 10. Dynamic host instructions reduced by the rules.

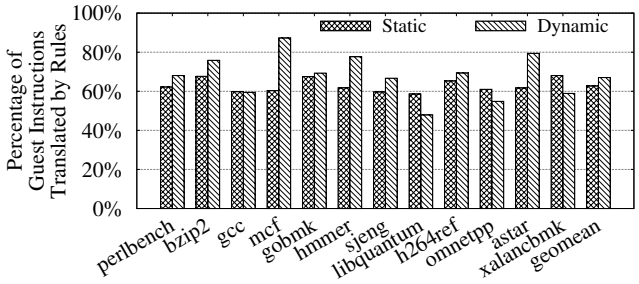


Figure 11. Static and dynamic coverage of translation rules.

translate the guest binaries produced by different compilers, Figure 9 shows the speedup achieved by our prototype and the implementation using LLVM JIT for guest binaries built by GCC. Again, the original QEMU is used as the baseline.

For the *test* workload, the translation overhead introduced by the rule-based translation is still negligible. For the *reference* workload, our prototype achieves a 1.21X speedup on average, which is again much better than that of LLVM JIT. This shows that our learning approach is quite insensitive to the compiler selected for learning the translation rules.

Figure 10 shows the percentage of the dynamic host instructions reduced by the learned rules using the *reference* workload compared to those executed in the original QEMU using the guest binaries built by LLVM. On average, 34% of the dynamically executed host instructions are reduced compared to the original QEMU. An interesting observation is that the percentage of dynamic host instructions reduced in *hmmer* is higher than that in *mcf* while the speedup achieved in *hmmer* is lower than that in *mcf* (see Figure 8). A possible reason could be the different types of dynamic host instructions (with different execution times) reduced by the learned rules in these two benchmarks.

Figure 10 also shows that the reduced percentages for some benchmarks are not very high, e.g., *omnetpp*. After looking into this application, we found that the five hottest basic blocks in this benchmark come from the LLVM runtime library, which is mainly composed of *hand-written* assembly code. This makes it hard for the rules learned from the source code of other benchmarks to cover such guest binary code.

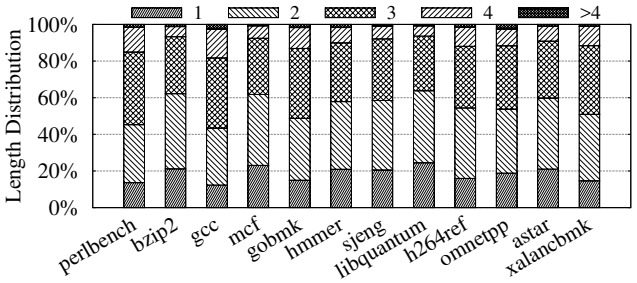


Figure 12. Length distribution of hit translation rules.

Assume we translate a total of m guest instructions during the emulation of an application p. The boolean flag B_i indicates if the ith guest instruction is translated by the learned translation rules. F_i represents the dynamic emulation frequency of the ith guest instruction. The static and dynamic coverage of the rules applied for p can be defined as $S_p = \frac{\sum B_i}{m}$ and $D_p = \frac{\sum F_i \times B_i}{\sum F_i}$. Figure 11 shows the experimental results of S_p and D_p for each benchmark. On average, our prototype can achieve more than 60% of both static and dynamic coverage. For *mcf*, the dynamic coverage is even higher than 85%. This demonstrates the effectiveness of the translation rules even though they are learned from significantly different applications.

The *length* of a translation rule is the number of guest instructions in this rule. A rule is a "hit" if there is at least one guest instruction sequence that matches this rule during the emulation of the evaluated benchmark. Figure 12 shows the length distribution of the hit rules applied on each benchmark. It is quite common to have hit rules with more than 2 guest instructions. Also, most of the rules are less than 6, as most lines in the source code are simple program statements.

7 Limitations and Discussion

Memory consistency models are very important for cross-ISA emulation, particularly for multi-threaded applications. Additional host memory fence instructions need to be inserted if the guest memory consistency model is stronger than the host memory consistency model. Some recent work has provided highly efficient and effective algorithms for such fence insertion [9, 23]. Our rule-based approach is only for the *translation phase* of a DBT system to translate a binary code from one ISA to another ISA. It is very challenging to do fence insertion during the translation phase because memory consistency models are not part of the instruction set architectures. They depend mostly on different *implementations* of the micro-architectures. Our learning approach is thus not targeting, but orthogonal to, the memory consistency issues. Fence insertion needs to be done after the translation phase using any of the above mentioned algorithms.

It is also well known that most compilers only use a very small subset of the ISA to generate binaries, which also holds

true in DBT systems. In our experimental study (as shown in Figure 11), our prototype can achieve around 60% of both static and dynamic coverage. Incomplete coverage is mostly due to insufficient translation rules for various combinations of guest instructions, which requires more training programs to build up the repertoire of such rules. To this end, the learning system could be trained using large amounts of existing open-source software. However, with more and more rules learned and accumulated, a more efficient management scheme is required to quickly search the matched rules during the binary translation. This will be an interesting issue that demands further investigation.

Due to the lack of hot trace generation support in QEMU, there is no trace optimization in our implementation. A trace can be generated by combining several basic blocks for further trace-level optimization across multiple blocks. However, as mentioned earlier, the learned rules are independent of QEMU, and thus trace-level optimization can be applied to its translated blocks to further improve their performance. One interesting issue though is the impact of the host codes obtained from the translated rules during the trace optimization as they are already more "optimized" than the translated codes from QEMU. This also warrants further studies.

8 Related Work

Bansal and Aiken [1] use peephole superoptimizers to generate translation rules for binary translation. The mechanism *exhaustively* enumerates *all* possible host instruction sequences up to a certain length and checks each of them to see if it is equivalent to *any* fixed-length guest instruction sequence. The equivalent guest and host instruction sequences can construct a translation rule. The generated rules are then fed to a *static* binary translator to generate more efficient host binary code.

There are three fundamental limitations in this approach. First, the process of generating translation rules is extremely time-consuming, especially for large existing ISAs such as x86, as it uses a close to brute force approach to search for such translation rules. Specifically, it took about one week to collect around 750 translation rules. Second, due to the substantial time needed to gather translation rules, the maximum lengths of guest and host instruction sequences are limited to 2 or 3, otherwise the time will increase exponentially. As shown in our results, it is very efficient and common to hit rules with more than 2 guest instructions. Third, it did not address the issues of how to integrate it to an existing DBT system that could benefit from the produced rules. The high translation overhead when applied to a static translation, i.e., around 2 ∼ 6 minutes, is infeasible to a DBT system. In contrast, our approach leverages the program source code to bridge the semantic gap between the guest and the host ISAs during the learning process, which makes our learning approach much more efficient, i.e., less than two seconds to learn a rule. Another main advantage of our learning approach is that it is *code-sequence* based, not *individual instruction* based, which allows more optimized code sequences to be generated for higher performance. In addition, we also address the implementation issues regarding how to extend an existing DBT system to benefit from the learned rules with very little additional performance overhead.

ISAMAP [31] leverages an architecture description language, ArchC, to establish instruction mappings between the guest and the host ISAs. However, there is only one guest instruction in each mapping, and it still requires human effort to construct the mappings. Compared with ISAMAP, our learning approach can automatically learn translation rules from program source code, and there is no length limitation on a guest code sequence in the learned rules.

HQEMU [16] relies on LLVM JIT to generate host binary code. The translation from QEMU's TCG Ops to LLVM IRs is still primarily based on manually constructed translation rules. As shown in our results, LLVM JIT introduces very heavy translation overhead. In contrast, the translation overhead incurred by the rule-based translation is very small even for short-running workloads.

The work in [14] automatically lifts assembly code to compiler IRs. However, the lifted IRs are still machine dependent, and thus cannot be applied to cross-ISA DBT systems. There has been a lot of research work to improve the performance of DBT systems [7, 9–11, 15, 17, 19, 32, 33, 35]. Our approach can typically take advantage of those techniques to further improve the performance of the generated host binaries.

9 Conclusion

In this paper, we present a novel approach to intelligently learn binary translation rules for DBT systems without manual intervention. It leverages the source programs to bridge the semantic gap between the guest and the host ISAs to produce binary translation rules very efficiently. These translation rules are verified using symbolic execution. A prototype has been implemented based on QEMU to demonstrate the benefits of the learned translation rules. Experimental results show that the average time of learning a translation rule can be less than two seconds. With the rules learned from all benchmarks in SPEC CINT2006 excluding the evaluated benchmark itself, we can achieve 1.25X speedup on average with very little translation overhead.

Acknowledgments

We are very grateful to David Grove and the anonymous reviewers for their valuable comments and feedback. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1514444. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] 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, USA, 177–192. http://dl.acm.org/citation.cfm?id=1855741.1855754

[2] 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, USA, 191–201. http://dl.acm.org/citation.cfm?id=956417.956550

[3] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference (ATC '05)*. USENIX Association, Berkeley, CA, USA, 41–46. http://dl.acm.org/citation.cfm?id=1247360.1247401

[4] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT – a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*. ACM, New York, NY, USA, 234–245. https://doi.org/10.1145/800027.808445

[5] Derek L. Bruening. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. Dissertation. Cambridge, MA, USA. AAI0807735.

[6] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 463–469. http://dl.acm.org/citation.cfm?id=2032305.2032342

[7] Chao-Jui Chang, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. 2014. Efficient Memory Virtualization for Cross-ISA System Mode Emulation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '14)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/2576195.2576201

[8] Eric Christopher. 2013. Debug Info - Status and Directions. In *European LLVM Conference*. http://llvm.org/devmtg/2013-04/christopher-slides.pdf

[9] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. 2017. Cross-ISA Machine Emulation for Multicores. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Computer Society, Piscataway, NJ, USA, 210–220. http://dl.acm.org/citation.cfm?id=3049832.3049855

[10] Amanieu D'Antras, Cosmin Gorgovan, Jim Garside, John Goodacre, and Mikel Luján. 2017. HyperMAMBO-X64: Using Virtualization to Support High-Performance Transparent Binary Translation. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '17)*. ACM, New York, NY, USA, 228–241. https://doi.org/10.1145/3050748.3050756

[11] Amanieu D'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. 2017. Low Overhead Dynamic Binary Translation on ARM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. ACM, New York, NY, USA, 333–346. https://doi.org/10.1145/3062341.3062371

[12] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-vectors and Arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*. Springer-Verlag, Berlin, Heidelberg, 519–531. http://dl.acm.org/citation.cfm?id=1770351.1770421

[13] GNU Project. Accessed: January 2018. The GNU Compiler Collection. (Accessed: January 2018). http://gcc.gnu.org

[14] Niranjan Hasabnis and R. Sekar. 2016. Lifting Assembly to Intermediate Representation: A Novel Approach Leveraging Compilers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 311–324. https://doi.org/10.1145/2872362.2872380

[15] Byron Hawkins, Brian Demsky, Derek Bruening, and Qin Zhao. 2015. Optimizing Binary Translation of Dynamically Generated Code. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society, Washington, DC, USA, 68–78. http://dl.acm.org/citation.cfm?id=2738600.2738610

[16] 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 Tenth International Symposium on Code Generation and Optimization (CGO '12)*. ACM, New York, NY, USA, 104–113. https://doi.org/10.1145/2259016.2259030

[17] Chun-Chen Hsu, Pangfeng Liu, Jan-Jan Wu, Pen-Chung Yew, Ding-Yong Hong, Wei-Chung Hsu, and Chien-Min Wang. 2013. Improving Dynamic Binary Optimization Through Early-exit Guided Code Region Formation. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '13)*. ACM, New York, NY, USA, 23–32. https://doi.org/10.1145/2451512.2451519

[18] Intel. 2003. *IA-32 Intel®Architecture Software Developer's Manual*.

[19] Ning Jia, Chun Yang, Jing Wang, Dong Tong, and Keyi Wang. 2013. SPIRE: Improving Dynamic Binary Translation Through SPC-indexed Indirect Branch Redirecting. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '13)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2451512.2451516

[20] Stephen Kyle, Igor Böhm, Björn Franke, Hugh Leather, and Nigel Topham. 2012. Efficiently Parallelizing Instruction Set Simulation of Embedded Multi-core Processors Using Region-based Just-in-time Dynamic Binary Translation. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES '12)*. ACM, New York, NY, USA, 21–30. https://doi.org/10.1145/2248418.2248422

[21] 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: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–88. http://dl.acm.org/citation.cfm?id=977395.977673

[22] 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, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[23] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. 2015. ArMOR: Defending Against Memory Consistency Model Mismatches in Heterogeneous Architectures. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 388–400. https://doi.org/10.1145/2749469.2750378

[24] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-exploration Lifting: Hi-fi Tests for Lo-fi Emulators. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 337–348. https://doi.org/10.1145/2150976.2151012

[25] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[26] Vijay Janapa Reddi, Dan Connors, Robert Cohn, and Michael D. Smith. 2007. Persistent Code Caching: Exploiting Code Reuse Across Executions and Applications. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*. IEEE Computer Society, Washington, DC, USA, 74–88. https://doi.org/10.1109/CGO.2007.29

[27] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 475–486. https://doi.org/10.1145/2485922.2485963

[28] David Seal. 2000. *ARM Architecture Reference Manual* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[29] Jim Smith and Ravi Nair. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[30] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS '08)*. Springer-Verlag, Berlin, Heidelberg, 1–25. https://doi.org/10.1007/978-3-540-89862-7_1

[31] Maxwell Souza, Daniel Nicácio, and Guido Araújo. 2012. ISAMAP: Instruction Mapping Driven by Dynamic Binary Translation. In *Computer Architecture: ISCA 2010 International Workshop AMAS-BT*. Springer Berlin Heidelberg, Berlin, Heidelberg, 117–138. https://doi.org/10.1007/978-3-642-24322-6_11

[32] Wenwen Wang, Chenggang Wu, Tongxin Bai, Zhenjiang Wang, Xiang Yuan, and Huimin Cui. 2014. A Pattern Translation Method for Flags in Binary Translation. *Journal of Computer Research and Development* 51, 10 (2014), 2336–2347. http://crad.ict.ac.cn/EN/10.7544/issn1000-1239.2014.20130018

[33] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. 2016. A General Persistent Code Caching Framework for Dynamic Binary Translation (DBT). In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16)*. USENIX Association, Berkeley, CA, USA, 591–603. http://dl.acm.org/citation.cfm?id=3026959.3027013

[34] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, Stephen McCamant, Youfeng Wu, and Jayaram Bobba. 2017. Enabling Cross-ISA Offloading for COTS Binaries. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '17)*. ACM, New York, NY, USA, 319–331. https://doi.org/10.1145/3081333.3081337

[35] Xiaochun Zhang, Qi Guo, Yunji Chen, Tianshi Chen, and Weiwu Hu. 2015. HERMES: A Fast cross-ISA Binary Translator with Post-optimization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society, Washington, DC, USA, 246–256. http://dl.acm.org/citation.cfm?id=2738600.2738631