Unleashing the Power of Learning: An Enhanced Learning-Based Approach for Dynamic Binary Translation

Changheng Song, *Fudan University;* Wenwen Wang, Pen-Chung Yew, and Antonia Zhai, *University of Minnesota;* Weihua Zhang, *Fudan University*

https://www.usenix.org/conference/atc19/presentation/song

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Unleashing the Power of Learning: An Enhanced Learning-Based Approach for Dynamic Binary Translation

Changheng Song[†], Wenwen Wang[‡], Pen-Chung Yew[‡], Antonia Zhai[‡], and Weihua Zhang[†]
[†]*Software School, Fudan University*
[†]*Shanghai Key Laboratory of Data Science, Fudan University*
[‡]*Department of Computer Science and Engineering, University of Minnesota, Twin Cities*
[†]{17212010032, zhangweihua}@fudan.edu.cn, [‡]{wang6495, yew, zhai}@umn.edu

Abstract

Dynamic binary translation (DBT) is a key system technology that enables many important system applications such as system virtualization and emulation. To achieve good performance, it is important for a DBT system to be equipped with high-quality translation rules. However, most translation rules in existing DBT systems are created manually with high engineering efforts and poor quality. To solve this problem, a learning-based approach was recently proposed to automatically learn semantically-equivalent translation rules, and symbolic verification is used to prove the semantic equivalence of such rules. But, they still suffer from some shortcomings.

In this paper, we first give an in-depth analysis on the constraints of prior learning-based methods and observe that the equivalence requirements are often unduly restrictive. It excludes many potentially high-quality rule candidates from being included and applied. Based on this observation, we propose an enhanced learning-based approach that relaxes such equivalence requirements but supplements them with constraining conditions to make them semantically equivalent when such rules are applied. Experimental results on SPEC CINT2006 show that the proposed approach can improve the dynamic coverage of the translation from 55.7% to 69.1% and the static coverage from 52.2% to 61.8%, compared to the original approach. Moreover, up to 1.65X performance speedup with an average of 1.19X are observed.

1 Introduction

Dynamic binary translation (DBT) is a key enabling technology for many critical system applications such as system virtualization and emulation [20, 28], whole program/system analysis [6, 13], software development and debugging [14], security vulnerability detection and defense [15, 17], computer architecture simulation [22, 27, 29], and mobile computation offloading [26]. There have been many widely-used DBT systems, such as Pin [18], Valgrind [21] and QEMU [2].

In general, a DBT system takes an executable binary code in one instruction set architecture (called *guest* ISA) and dynamically translates it into the binary code in another instruction set architecture (called *host* ISA). The translation process is mostly driven by *translation rules* that translate guest instructions into a sequence of semantically-equivalent host instructions [23].

For a DBT system, its performance is dominated by the quality of the translated host binary code [25]. Therefore, it is very important for a DBT system to be equipped with high-quality translation rules. However, due to the complexity and opacity of modern ISAs, it is difficult to *manually* construct such high-quality translation rules as it poses a significant engineering challenge. Even worse, to support retargetable DBTs (from multiple guest ISAs into multiple host ISAs) in the same framework, a set of pseudo-instructions are commonly used as their internal representations [2]. As the execution time is directly proportionate to the number of host instructions executed, such a multiplying effect has a significant impact on the overall DBT performance.

To improve the quality of translation rules and reduce engineering efforts, a learning-based approach [23] is recently proposed to learn automatically binary translation rules. Since the translation rules are learned from the optimized binary codes generated by the compiler, this approach is capable of yielding higher quality translation rules than existing manual schemes. Moreover, the whole learning process can be fully automated without manual intervention. Although the above approach is attractive, it still suffers from some fundamental limitations. That is, a translation rule can be harvested (i.e., learned) only if the guest and the host binary code that correspond to the same program source statement(s) are strictly semantically equivalent. This is enforced through a symbolic verification process.

On the surface, this equivalence verification process is necessary and appropriate because it guarantees the *correctness* of the learned rules. However, further investigation reveals that this equivalence requirement is often unduly restrictive. It excludes many potentially high-quality rule candidates from being harvested and applied. In particular, such restrictions usually keep architecture-specific instructions in guest and/or host ISAs from being included for more efficient translation as they are mostly architecturally specific, and thus inherently different and more challenging to prove semantic equivalence.

To overcome this limitation, this paper presents an enhanced learning-based approach that relaxes such restrictions and allows more translation rules to be harvested and applied. More specifically, it purposely relaxes the requirements of semantic equivalence and allows semantic discrepancies between the guest and host instructions to exist in the translation rules, e.g., different condition codes or the different number of operands in matching guest and host instructions. Symbolic verification process is no longer just to check the *strict* semantic equivalence between the matching guest and host instructions, but also to identify the specific semantic discrepancies between them that can be used during the rule application phase to verify whether such discrepancies either will not cause ill effect, or are satisfied in the context of the rules being applied (for more details see Section 4). We call such semantic equivalence in the translation rules *constrained semantic equivalence* as the specific semantic discrepancies of the translation rules become the *constraining conditions* for such rules to be safely applied. This requires some runtime program analysis (mostly in a very limited scope) during the rule application phase, which usually incurs very small overhead. Those with very complicated constraining conditions that require extensive runtime program analysis will be discarded.

To demonstrate the feasibility and the benefit of such constrained-equivalent translation rules, we have implemented a prototype based on the proposed approach. The prototype includes an enhanced learning framework and a DBT system that applies the constrained-equivalent translation rules to generate host binary code. We evaluate the implemented prototype using SPEC CINT2006. Experimental result shows that the proposed approach can significantly improve the harvest rate of the learning process from 20.3% to 25.1% and dynamic coverage from 55.7% to 69.1% while static coverage from 52.2% to 61.8%, compared to the original learning approach in [23]. Moreover, no degradation is observed for the learning efficiency, i.e., around 2 seconds to yield a translation rule, which is the same as the original learning process. After applying the enhanced translation rules, we achieve up to 1.65X performance speedup with an average of 1.19X compared to the original approach.

In summary, this paper makes the following contributions:

- We propose an enhanced learning-based approach that can harvest and apply constrained-equivalent translation rules discarded by the original approach, and allows DBT systems to generate more efficient host binary code.

- We implement the proposed learning-based approach in a prototype, which includes a learning framework based on LLVM and a DBT system extended from QEMU to accept the constrained-equivalent translation rules.

- We conduct some experiments to evaluate the proposed learning-based approach. Experimental results on SPEC CINT2006 shows that our approach can achieve up to 1.65X speedup with an average of 1.19X compared to the original learning approach.

The rest of this paper is organized as follows. Section 2 presents some background of the original non-constrained semantically-equivalent learning-based approach. In Section 3, we identify some technical challenges in learning and applying constrained-equivalent translation rules. Section 4 presents the design issues of our enhanced learning-based approach. In Section 5, we describe some implementation details of the prototype and evaluate the proposed approach and show some experimental results. Section 6 presents some related work and Section 7 concludes the paper.

2 Background

In this section, we introduce some background information on how a DBT and a learning-based approach such as the one proposed in [23] work.

2.1 Dynamic Binary Translation (DBT)

Typically, a DBT system adopts a guest *basic block* (or *block* for short) as the translation unit to translate guest binary code into host binary code. A basic block comprises a sequence of instructions with only one entry and one exit, and thus whenever the first instruction of a basic block is executed, the rest of the instructions in this block will be executed exactly once in order. It is worth noting that, due to the semantic differences between the guest and host ISAs, one guest block may be translated into multiple host blocks by the DBT system.

To translate a guest basic block, the DBT system firstly disassembles the guest binaries to obtain guest assembly instructions. Then, it tries to match the guest instructions with available *translation rules*. After a matched translation rule is found, the corresponding guest instructions are translated into host instructions as specified in the translation rule. This process could be iterated multiple times until all instructions in the guest block are translated. Finally, the generated host instructions are assembled into host binaries and executed directly on host machines. Figure 1 shows an example of such a translation process, where ARM is the guest ISA, and x86 is the host ISA. In this example, two translation rules are applied to translate two ARM instructions into two x86 instructions, respectively.

To mitigate the performance overhead incurred during the translation process, especially for short-running guest applications, the translated host binary code is stored into a memory region called *code cache*, and reused in the later execution. After all instructions in a guest block are translated, the execution flow of the DBT system is transferred to the code cache.

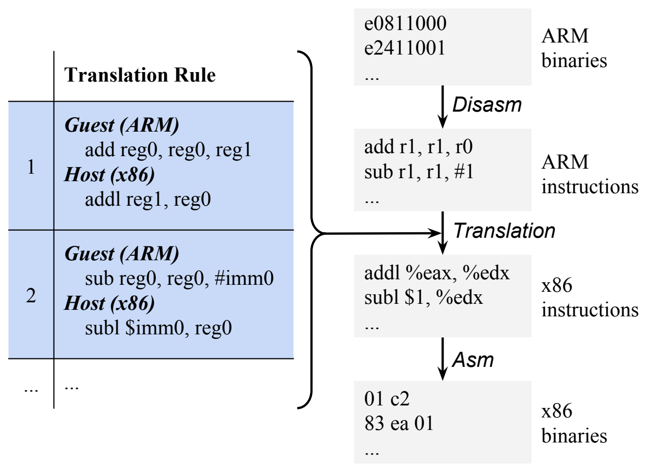


Figure 1: Dynamic binary translation from ARM to x86 driven by manually-constructed translation rules. Here, for simplicity, we assume the guest registers `r0` and `r1` are emulated using the host registers `eax` and `edx`, respectively.

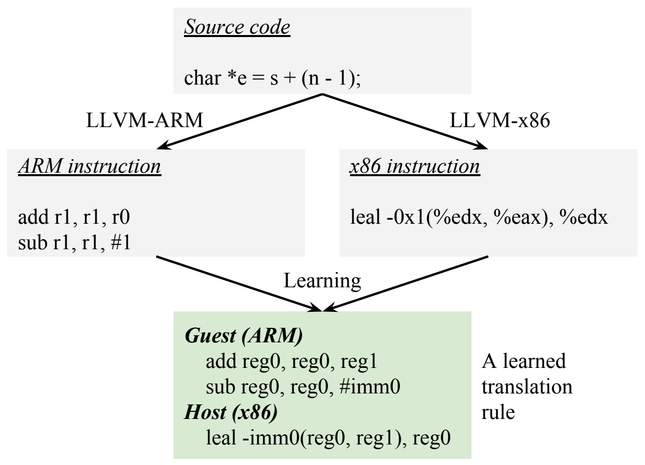


Figure 2: Automatically learning binary translation rules during the compilation process of program source code. Compared to the translation rules used in Figure 1, the learned translation rule can generate more efficient host binary code.

A *hash table* is employed to establish the mapping between the guest binary code and the corresponding translated host binary code in the code cache. Each time a guest block is encountered, the hash table is looked up to find out whether there exists a host code in the code cache that corresponds to this guest block. If yes, the translation process will be skipped, and the stored host binary code will be executed. Otherwise, the guest block is translated, and the hash table is updated with the added translated host binary.

2.2 Learning Translation Rules

As mentioned earlier, the translation process in a DBT system is mainly directed by translation rules, which also determine the quality (i.e., performance) of the translated host binary code. Therefore, it is vital for a DBT system to have high-quality translation rules for better performance. However, in practice, it is a significant engineering challenge to develop high-quality translation rules as most translation rules in existing DBT systems are constructed manually by developers. Moreover, modern ISAs are often documented in obscure and tediously long manuals. For example, Intel's manual has around 1500 pages for the x86 ISA. It requires substantial engineering efforts to understand both the guest and the host ISAs to construct high-quality translation rules.

To solve this problem, a recent approach proposes to automatically learn binary translation rules [23]. More specifically, this approach uses the same compiler for different ISAs, i.e., LLVM-ARM and LLVM-x86, to compile the same source program. During the compilation process, it extracts binary translation rules from ARM and x86 binary code that correspond to the same program source statement(s). This is inspired by the observation that the binary code compiled by the same compiler for different ISAs from the same program source code should be equivalent in program semantics. To further enforce such equivalence requirement, a symbolic verification engine is developed to filter out rule candidates in which guest and host binary code are not semantically equivalent.

Figure 2 illustrates an example of the above learning process, In this example, the program source statement is compiled into two ARM instructions and one x86 instruction by LLVM-ARM and LLVM-x86, respectively. Using symbolic execution, we can verify that the guest ARM register `r1` and the host x86 register `edx` should have the same value assuming the same initial condition. We can thus prove that the sequence of the two ARM instructions is semantically equivalent to the single x86 instruction in the example. A translation rule that maps the sequence of the two ARM instructions into one x86 instruction can then be harvested. Recall the example in Figure 1. If we use this learned rule to translate the guest binaries, we only need one host instruction instead of two as shown in the example, i.e. more efficient host binary code can be generated.

3 Issues and Challenges

The significance of the above learning approach is two folds. Firstly, it can automatically learn binary translation rules for DBT systems with less burden on developers. Secondly, given that the translation rules are learned directly from binary code generated by the native compilers, it is more likely that the harvested translation rules are more optimized than the translation rules naïvely constructed by hand, as shown in Figure 2 and Figure 1.

Theoretically, if we keep training such a learning-based

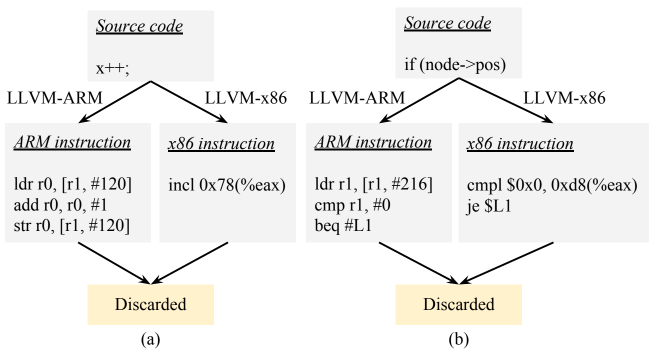


Figure 3: Two examples to demonstrate the limitation of the learning approach in [23]. These two rule candidates are discarded because the guest registers `r0` in (a) and `r1` in (b) have no equivalent host register.

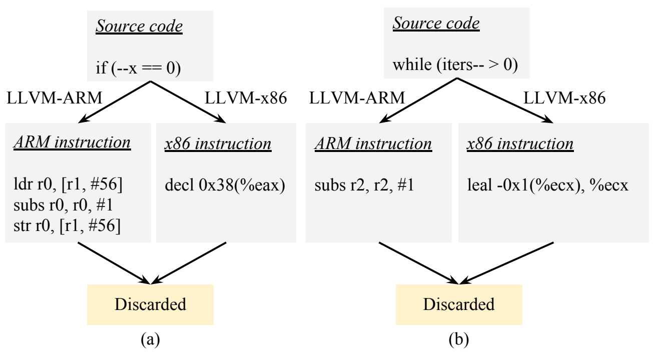


Figure 4: Another two rule candidates discarded by the learning approach in [23] because of the different condition codes in guest and host ISAs.

system with a large number of source programs, we should be able to harvest a large number of translation rules and apply them to guest binaries with good coverage. Unfortunately, after a more thorough study of this approach, we found it suffers from a fundamental limitation that prohibits it from harvesting many high-quality translation rules. In this section, we explain in more details such limitations and identify some technical challenges if we want to overcome them.

A Fundamental Limitation. To guarantee the correctness of the learned translation rules, it employs a symbolic verification engine to check the exact semantic equivalence between the guest and host binary code sequences. More specifically, the semantic equivalence is verified in three aspects that include matching register operands, memory operands and branch conditions. More details can be found in [23].

If the verification results show that the guest and host binary code sequences are not strictly equivalent, the rule candidate is discarded and no translation rule is harvested. Undoubtedly, such a verification process is necessary and appropriate. However, by a more detailed study on the discarded rule candidates, we found that the requirement of *exact* semantic equivalence is too restrictive. Many high-quality rule candidates are forced to be discarded, especially those guest and host binary code sequences that are more architecturally specific and their ISAs are significantly different, such as ARM (a reduced instruction set computer (RISC)) and Intel x86 (a complex instruction set computer (CISC)) in our example.

Figure 3 shows two examples of this limitation. Here, similar to the previous examples, the guest ISA is ARM and the host ISA is Intel x86. In Figure 3(a), the value of the variable `x` is increased by one through the increment operator as shown in the source code. With its RISC ISA, the ARM compiler generates three instructions for this source statement: loading the original value of `x`, performing the addition, and then storing the result back to `x`. In contrast, the Intel x86 compiler needs only one instruction, `incl`, with its CISC ISA.

Similarly, in Figure 3(b), the x86 instruction `cmpl` can have a memory operand, but an ARM `ldr` instruction is required before the `cmp` instruction. In these two cases, the verification will fail because there is a mismatch of register operands between the guest and the host code sequences, i.e. there is no host register that matches and holds the same value as the guest register `r0`.

However, if we examine these two examples more carefully, we will find that the root cause of the failed verification stems from the inherent differences between the guest and the host ISAs. In practice, such architectural differences are quite common and pervasive in different ISAs, even if they are both RISCs or CISCs. For instance, a post-indexed memory load instruction in ARM will modify the address register after the loading operation, while there is no similar instruction in MIPS, which is another representative RISC ISA.

In fact, these differences represent the essence of the architectural design unique to each ISAs. It is indeed a huge loss for a learning-based approach to discard such rule candidates simply because of their ISA differences. As they are architecturally specific, they are often the most efficient code sequences selected by the native compilers for specific program contexts and thus have a high potential to turn into high-quality translation rules.

Another shortcoming resulted from the aforementioned limitation is that it also excludes many rule candidates that contain instructions associated with architecture-specific hardware support. For instance, many architectures have condition codes (also known as *eflags* in x86 machines). They are single-bit registers used to store the execution summary of an instruction and can influence the control flow of the later instructions. In particular, ARM has four condition codes: negative (NF), zero (ZF), carry (CF), and overflow (VF), while x86 has seven condition codes: carry (CF), parity (PF), adjust (AF), zero (ZF), sign (SF), direction (DF), and overflow (OF).

Figure 4 shows two examples with instructions related to condition codes. In Figure 4(a), the source code decreases the value of `x` by one and then checks the result to see whether it is zero or not. An ARM instruction `subs` is generated to perform the subtraction and update the condition codes. Here, `subs` updates all four ARM condition codes, including CF. Similarly, an x86 instruction `decl` is used to decrease the value stored in the memory operand by one and update the condition codes. However, `decl` updates all x86 condition codes, except CF. As a result, the verification process in the original learning-based approach will consider the ARM and x86 code are not semantically equivalent and discard this rule candidate. Similarly, the rule candidate in Figure 4(b) is also discarded because the x86 instruction `leal` does not update any condition code. In fact, the source code in Figure 4(a) only needs to check whether the result is zero or not, which only requires the condition code ZF. Thus, it is unnecessary to update the condition code CF, as it is never used in this context. That means, it is still possible to harvest this translation rule and apply it, if the ARM condition code CF is not used (i.e. *dead*) in the later code before it is updated. Similarly, the rule candidate in Figure 4(b) can also be harvested.

Technical Challenges. Although such limitations could exclude many high-quality translation rules during the learning process, it faces several technical challenges if we want to harvest them and apply them in a DBT system for a better performance and higher coverage.

First, we have to relax the original verification objectives as they are designed to verify the *exact* equivalence between the guest and host code sequences.

Second, given that most of those translation rules are not strictly equivalent, it is imperative that we have a mechanism to enforce their correctness when we apply them. Equally important is that the performance overhead incurred by such enforcement should be less than the performance gain they can provide.

Last but not least, in the original learning approach, a learned translation rule only needs to include two parts, i.e., the guest and host instructions, and this is typically sufficient for a DBT system. However, for the constrained-equivalent translation rules, whether we can apply these rules at runtime or not depends on the context they are being applied. As a result, we need to extend the structure of translation rules to include such constraining requirements.

4 An Enhanced Learning-Based Approach

In this section, we present the design of the proposed enhanced learning-based scheme, starting with an overview of the system framework.

4.1 Overview

The major goal of our enhanced learning-based approach is to learn and apply high-quality translation rules excluded by the original learning approach. These translation rules contain constrained-equivalent guest and host instructions, and thus cannot be harvested using the original learning approach. To this end, we redesign the learning process, reorganize the structure of the learned translation rules, and make necessary extensions to the DBT system to allow the application of the constrained-equivalent translation rules.

Figure 5 illustrates the workflow of our enhanced learning-based approach. To learn translation rules, we also compile the same program source code using the same compiler for guest and host ISAs to generate two versions of the binary code. We then extract guest and host code sequences that correspond to the same *learning scope* and consider them as the candidates for the translation rules. The learning scope is defined at the program source code level. In the original learning system, the default learning scope is set to be one source statement. The extracted guest and host code sequences then form a *rule candidate*. For each rule candidate, the next step is to verify whether the corresponding guest and host code sequences are *constrained equivalents* or not. If yes, a translation rule can be harvested. Otherwise, the rule candidate is discarded.

As an example to demonstrate our approach and by studying the rule candidates discarded by the original learning scheme, we consider the guest and host code sequences in a rule candidate as *constrained equivalent* if every modified guest *storage operand* contains the same value as a modified host storage location at the end of the code sequences and vice versa. Here the *storage operand* is broadly defined, as it can be either a register, a memory location, or a condition code (i.e., eflag). Furthermore, it is allowed that there is no corresponding modified storage location in the *host* code sequences, e.g., a corresponding condition code as mentioned earlier.

Using this relaxed and constrained equivalence definition, the guest and host code sequences can be semantically equivalent only if all *modified* guest storage operands without the corresponding host storage operands (e.g., condition codes) are not used in the following guest binaries before they are modified again. These modified guest storage operands without the corresponding host storage operands can be considered as the *constraining condition* of this constrained-equivalent translation rule.

In our framework, the semantic equivalence can be relaxed in other ways as long as the *discrepancies* can be identified and shown either having no ill effect in the context they are applied or can be compensated to make them semantically equivalent when they are applied. In other words, their *constraining conditions* can be identified and satisfied when these rules are applied. To simplify our prototype design, we only consider relaxing the requirement of exact mapping of the storage operands as defined earlier. The identified constraining conditions are integrated into the learned translation rules to determine whether it is safe to apply them or not. It is worth noting that for strictly equivalent translation rules the constraining condition is null.

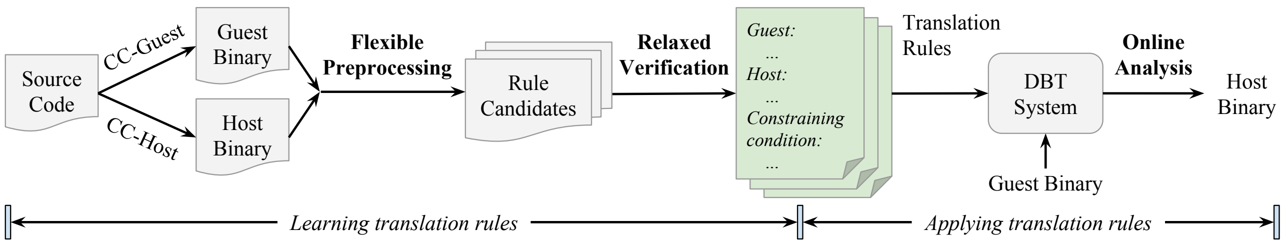


Figure 5: The work flow of the proposed enhanced learning-based approach.

To determine whether the constraining condition is met or not, a lightweight runtime analysis is employed to determine the program context in which the guest instructions is to be translated. In our case, the program context includes the information about which guest storage operand is modified by the guest instructions and used in the later code before it is modified again. The program context is then used to verify whether the constraining condition is satisfied or not. If yes, the translation rule can be applied, otherwise, it is discarded.

4.2 Varying Learning Scopes

In the original learning scheme, the learning scope is limited to one source statement. Although it appears to be reasonable, it may miss potential rule candidates as it is very common for compilers to perform optimization across multiple source statements. Therefore, our enhanced learning approach varies the learning scope from one to n source statements, and apply each learning scope over the source program.

More specifically, a *sliding window* is employed. The sliding window of size n covers n contiguous source statements, i.e. the sliding window covers the learning scope of n statements. Guest and host instructions that correspond to the n statements in this window are extracted as a rule candidate. The sliding window moves from the first line of the source code toward the end of the code. The window size is initially set to 1, and incremented by one after each pass through the sources code. When the window of size i is moved through all the source code, the number of rules learned from current window size will be compared to the number of rules learned from window sizes 1 to $i-1$. If new rules learned from window size i are less than 10% of all learned rules from window sizes 1 to $i-1$, the learning process will be stopped.

4.3 Learning Constrained-Equivalent Rules

To verify the constrained equivalence of the guest and host instructions in a rule candidate, we use the same symbolic verification engine, but relax the requirements for semantic equivalence.

First, we establish an *initial mapping* between guest and host live-in operands the same way as the original learning approach, i.e., guest registers $\mapsto$ host registers, guest memories $\mapsto$ host memories, and guest immediate values $\mapsto$ host immediate values (i.e. constants). Then, we initialize the mapped guest and host operands with the same symbol values and symbolically execute the guest and host code sequences, respectively. After the symbolic execution, we extract the symbol results of the modified guest and host registers, memories, and condition codes. These results are then fed into a SMT solver to figure out, for each modified guest register/memory/condition code, whether there exists a modified host register/memory/condition code corresponding to it or not. If each modified guest operand is mapped to a modified host operand, an original rule is generated.

If the SMT solver indicates that there exists a modified guest memory operand that does not have a matching host memory operand, we discard this rule candidate. If a modified guest register/condition code has no matching modified host register/condition code, we can harvest this rule candidate as the guest and host instructions can still be constrained equivalent. Moreover, such unmatched guest registers/condition codes are recorded as constraining conditions of the learned rules and will be checked when the rules are applied. The reason for discarding candidate rules with unmatched memory operands is that the resulting constraining conditions will require time-consuming data dependence analysis to determine whether the constrained equivalence is satisfied or not when such rules are applied.

Otherwise, all other rules are considered as a non-equivalent rule and be discarded.

4.4 Lightweight Online Analysis

For each constrained-equivalent rule to be applied, an online analysis is invoked to analyze the program context of the guest code sequence. The context information includes the data flow of the guest registers and condition codes, which can be obtained by statically analyzing the guest instructions. The context information is then used to determine whether the constraining condition of the matched constrained-equivalent rule is satisfied. For instance, if the analysis shows that a modified guest register is not mapped to any modified host register in the rule, and this modified guest register is not used in the following guest code, we can determine that the constraining condition has been satisfied in the program context, and the translation rule can be applied.

In general, to collect the context information, the online analysis examines guest instructions that are executed after the matched guest instruction sequence. Each instruction is examined to see whether it defines or uses the register(s) or condition code(s) specified in the constraining condition of the matched translation rule. If a definition can be found before usage on *all* paths following the matched guest code sequence, the matched rule can be applied safely. Otherwise, if usage is found, the matched rule should not be applied as the modified guest register/condition code is used but the matched rule does not update it.

For indirect branch instructions, it is quite difficult to identify all possible branch targets statically. For simplicity, we stop the online analysis when an indirect branch is encountered and the translation rule will not be applied for safety consideration.

4.5 Handling Predicated Instructions

Predicated instructions are very common in many ISAs, e.g., ARM and MIPS. A predicated instruction executes only if its predicate bit is "True". Otherwise, the instruction is a "nop". For example, "add ne r0, r0, r1" in ARM will be executed only when the condition code is not equal ("ne"). Some ISAs like x86 do not support predication, and conditional branches are used instead. It is worth noting that the original learning approach cannot handle predicated instructions. So how to efficiently support predicated instructions is another important design issue for a learning-based approach because the predicate tag in predicated instructions and conditional branch are not equivalent although the execution results are the same.

In translation, we use a lightweight analysis to divide predicated instructions into multiple blocks and generate conditional branches around those blocks according to their predicate information to support the translation of predicate instructions. Before translating a basic block, we first check the predicated condition of all instructions and divide the basic block into multiple *condition blocks*. Each condition block includes instructions with the same predicated condition. In one condition block, the translation rules can be directly applied without considering the predicated condition. After a condition block is translated, a branch instruction with the opposite condition is added to the host basic block before the translated condition block is added to the host block. The branch target is the instruction following the end of the host block. This analysis is very lightweight and each basic block only needs to be checked once.

Note that an instruction with a predicated condition may change the condition codes itself. For example, `cmp ne r0, 0` will update the condition code if the last condition code is *not equal*. So, instructions after these instructions that may change condition codes should be divided into a new condition block even the predicated condition is the same.

4.6 Discussion

Our enhanced learning approach currently only supports user-level applications. The translation for full-system level applications is not supported because full-system translation is more complex with mechanisms such as system calls, interrupts and device I/O. These mechanisms make the learning and matching of rules more difficult. It is left in our future work.

ABIs and many instructions such as *indirect branches* are not supported either. For ABIs, the calling conventions, such as how parameters are passed and how many parameters are used, are difficult to be identified and translated by rules. For example, ARM use registers to pass parameters but no specific instructions are used. But in X86, the *push* instructions will be used for passing parameters. For indirect branches, DBT systems usually search the branch target address according to a branch table maintained at runtime, which is not available at compile time. It makes it impossible to translate by our learned rules.

5 Experimental Results

In this section, we evaluate our prototype and address the following research questions:

1. How much performance improvement can be obtained by our enhance learning scheme in which we relax the requirement of matching storage operands as described in Section 4?
2. Where does the performance improvement come from when we include the added constrained-equivalent translation rules?
3. What is the effect of relaxing the strict semantic equivalence requirement?
4. How much overhead will the dynamic analysis incur?

5.1 Experimental Setup

Our enhanced learning-based DBT prototype is implemented based on QEMU (version 2.6.0) which is the same as the original learning scheme. The guest ISA is ARM, and the host ISA is x86. The LLVM compiler (version 3.8.0) is used to generate binary code for guest/host ISAs. All binary codes are generated using the same optimization level -O2. The same version of source code and guest/host binary code are used for comparison. One machine with 3.33GHz Intel Core i7-980x with six cores (12 threads) and 16GB memory is set up exclusively for performance evaluation. The operating system is the 32-bit Ubuntu 14.04 with Linux 3.13 for both machines. We used an older version of the system because we need to compare our new approach with the original approach, which used the same older version of the system. Besides, our

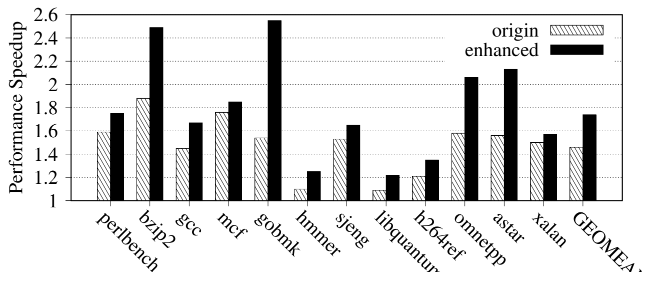


Figure 6: Performance comparison of the original and enhanced learning-based approaches.

experimental results are valid regardless of the system version used.

We use the 12 benchmarks in SPEC CPU INT 2006 with the reference inputs in our studies. To be as close to the real-world usage scenarios as possible, the performance of each benchmark is evaluated by the rules learned from other 11 benchmarks excluding evaluated benchmark itself. Each benchmark is run three times to reduce the random influence. The enhanced learning scheme described earlier is implemented in Python. The enhanced verification is implemented based on FuzzBALL [19], a symbolic execution engine. The tiny code generator (TCG) in QEMU is also enhanced to support the translation of predicated instructions as described in the last subsection, As mentioned earlier, to apply the constrained-equivalent translation rules, the dynamic analysis should be performed before such rules are applied.

5.2 Performance Results

Figure 6 shows the performance comparison of our enhanced learning-based scheme (marked as *enhanced*) and the original learning-based approach (marked as *origin*). The performance of QEMU without using any of the learning schemes is used as the baseline. Table 1 shows the MIPS of performance of original and enhanced approaches.

Using the *ref* input for all SPEC benchmarks, the quality of the translated host code is the major factor that impacts the overall performance. As shown in Figure 6, our enhanced learning scheme can achieve a performance improvement of up to 2.55X with an average of 1.74X compared to QEMU, which is a 1.19X improvement over the original learning approach on average.

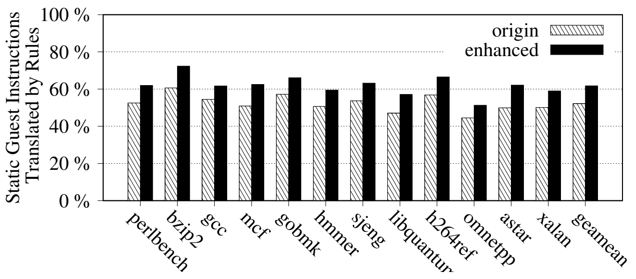
By studying the learned translation rules and how they are applied using our enhanced learning approach, we have the following observations on how they impact the overall performance. First, constrained-equivalent translation rules can usually be applied quite successful. The modified guest registers/condition codes that have no matching modified host registers/condition codes will usually be modified quickly again. This means they are only used to hold temporary value as we expected. Hence, relaxing strict matching requirements for storage operands can yield more translation rules, albeit constrained-equivalent rules, and can be applied quite effectively.

Table 1: MIPS of the original and enhanced learning-based approaches

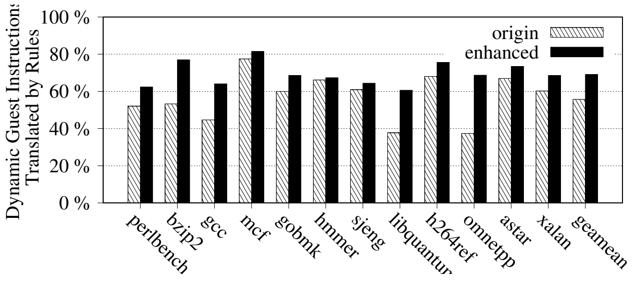
Benchmarks	Original	enhanced
perlbench	221.63	250.15
bzip2	1211.32	1388.92
gcc	521.78	575.09
mcf	603.99	739.93
gobmk	372.18	616.31
hmmer	1448.56	1632.93
sjeng	474.71	485.23
libquantum	1469.59	1532.97
h264ref	189.99	215.69
omnetpp	195.28	284.74
astar	396.10	562.51
xalan	283.19	290.52
GEOMEAN	475.15	567.29

Second, by relaxing the equivalence constraints to allow constrained-equivalent translation rules that can include predicated and condition instructions greatly improve the overall performance. This is because typical DBTs such as QEMU usually use memory locations to emulate the condition codes. Such an approach will incur many additional memory operations to access and update those condition codes in memory and incur very high overhead. But the constrained-equivalent translation rules can take advantage of the host condition codes to emulate guest condition codes, which can significantly reduce such overheads.

Figure 7(a) and Figure 7(b) show the static and dynamic coverage of the guest binaries using the origin and our enhanced learning-based schemes, respectively. The *"Coverage"* here is defined as the percentage of guest instructions that can be translated by the learned rules. So the *"static"* coverage is the percentage of static code translated by learned rules and *"dynamic"* coverage here is the percentage of *"executed"* guest instructions translated by learned rules. Compare to the original learning-based scheme, our enhanced learning scheme can improve the static coverage from 52.2% to 61.8%, and the dynamic coverage from 55.7% to 69.1% on average. It is worth noting that *gcc* and *libquantum* have a much higher dynamic coverage improvement than others, but do not get an expected higher performance improvement. Conversely, *gobmk* attains a high performance improvement but not as much coverage improvement. The reason is that many high-quality rules are applied when translating *gobmk*, but in *gcc* and *libquantum*, the applied rules can only attain moderate improvement. This seems to indicate that the coverage improvement does not translate directly to the overall performance improvement, but could be an important secondary effect.

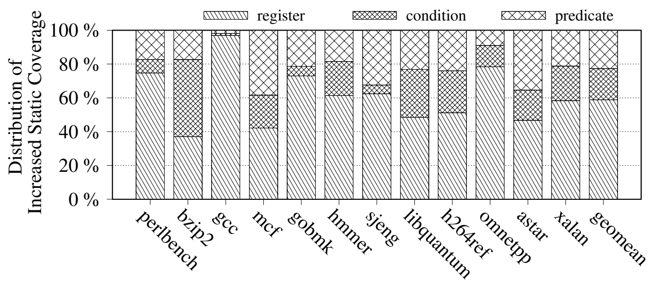


(a) Static coverage

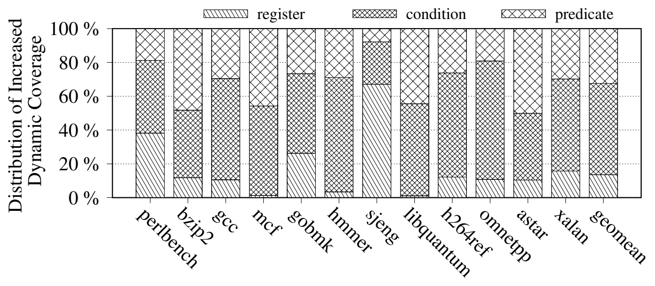


(b) Dynamic coverage

Figure 7: Static and dynamic coverage of translation rules.



(a) Distribution of increased static coverage



(b) Distribution of increased dynamic coverage

Figure 8: Distribution of the improved rule coverage.

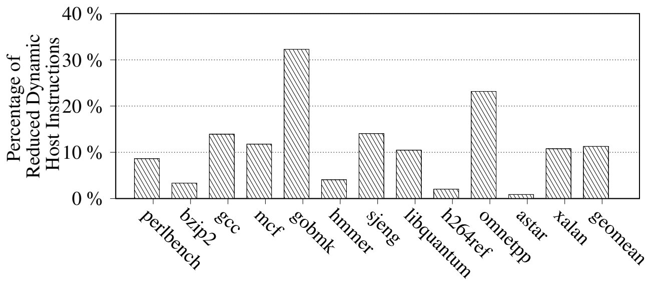


Figure 9: Reduction in dynamic host instruction counts by enhanced learning and translation.

To address the question of "where does the performance improvement come from when we include the added constrained-equivalent translation rules?", we analyze the coverage of the added translation rules when they are applied in each program. The results are shown in Figure 8. As described in Section 4, there are three major components in our relaxed equivalence constraints, i.e. we remove the strict requirement of exact matching. They are (1) register operands (marked as *register*), (2) condition-code operands (marked as *condition*), and (3) predicated-related instructions (marked as *predicate*). We did not include memory operands because they require more complicated data dependence analysis when they are applied.

A very interesting observation is that, among the added constrained-equivalent translation rules (their increased coverage is shown in Figure 7), the register-related constrained-equivalent translation rules constitute 58.83% of the static instructions on average. However, they only constitute 13.58% of added dynamic coverage. But the dynamic coverage of condition-code related rules is increased to 54.02% on average, while their static coverage is only 18.52%. This is because the condition codes are usually associated with the bound check, such as at the end of a loop. So, these instructions will be executed more frequently than others in their dynamic coverage.

To study the quality/efficiency of translated rules, Figure 9 shows the percentages of the reduced host instructions. On average, our enhanced learning scheme can reduce 11.28% of the total dynamic host instructions compared to the original learning scheme. We observe that the reduction in the host instructions of *gobmk* is higher than 30%, and in *omnetpp*, it is higher than 20%. However, in *gcc* and *libquantum*, the reduction is only about 10%. This also confirm our observation that rules applied in *gobmk* and *omentpp* translation have a higher quality, i.e. fewer host instructions in those translation rules, than rules in *gcc* and *libquantum*. But, we also notice that *bzip2* and *astar* attain a high performance improvement but only a moderate number of host instructions are reduced. One probable explanation is that even though they may have similar dynamic host instruction counts, more efficient and architecture-specific host instructions may have been used.

5.3 Learning Results

We further study the effect of our proposed relaxed learning scheme in other related aspects. The first is about the "yield" obtained during the learning phase, which shows how

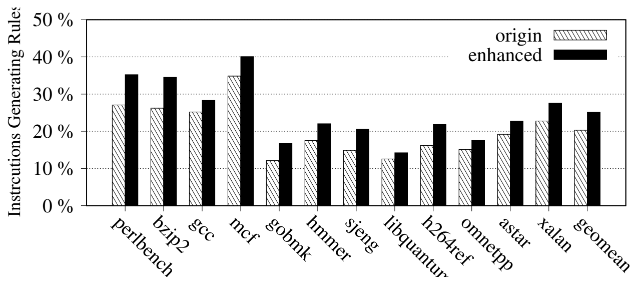


Figure 10: The yields of rule learning.

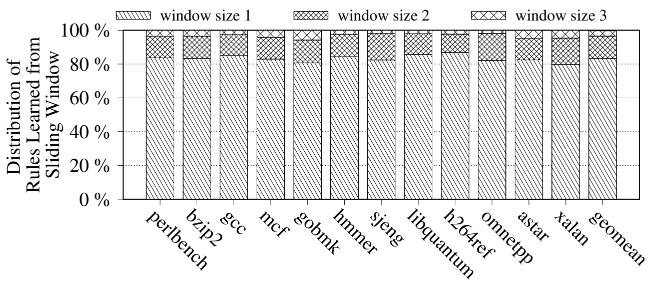


Figure 11: Rules distribution of sliding windows.

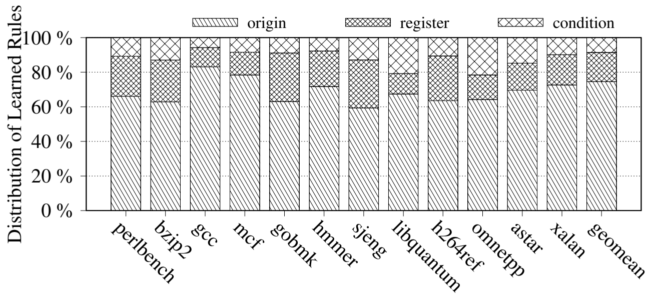


Figure 12: Distribution of rules learned by enhanced learning.

many translation rules can be harvested among the candidate rules during the learning phase. The other aspects include the effect of using the sliding window, and the distribution of the constrained-equivalent rules learned based on our relaxed equivalence constraints, i.e. relaxing the strict matching requirement on three storage operand types described in Section 4.

Figure 10 shows the yield obtained during the learning phase using the original learning scheme (marked as *origin*) and our enhanced learning scheme (marked as *enhanced*). The learning yield is increased from 20.3%to 25.1%. Even though the improvement in learning yield is moderate as we only made moderate relaxation on the semantic equivalence requirements, but as the obtained performance results show the quality of these rules is quite high. The moderate yield improvement also shows that there is a high potential for more high-quality rules to be learned and harvested. Another interesting question is that if a significant number of more source programs is used in the training phase, even with a low yield, how much more rules can be learned, and how much more performance improvement can be achieved by applying those added rules. These questions are beyond the scope of this paper.

Figure 11 shows the distribution of the translation rules learned using a flexible sliding window. We only show the data for a window size of up to three source statements because significantly fewer rules can be learned beyond 3 source statements. As the result shows, 13.16% of new rules can be learned from a window size of 2 and 3 source statements. The rules learned from window size 3 and beyond are less than 3.31%. So, a larger learning window is not necessary.

Figure 12 shows the distribution of the rules we learned using our enhanced learning scheme. On average, 16.84% of the learned rules are register-related (marked as *register*), while 8.16% are condition-code related (marked as *condition*) rules. We find that many constrained-equivalent rules related to local registers are used to load values from the memory before some computation, and are stored back to the memory after the computation. This is because RISCs are primarily "load/store" architectures, i.e. values in memory must be loaded into registers before computation and stored back to memory when the computation is completed. So many temporary/local registers are used. Another observation is that the amount of register reuse is minimal on RISCs, only 3.57% in total. So, only in rare situations, the compiler will use multiple registers instead of only one register. Such behavior is reflected in the use of the learned rules in the application phase.

5.4 Performance Overhead of Online Analysis

As the lightweight dynamic analysis is needed in the application of the constrained-equivalent translation rules, its runtime overhead needs to be evaluated. Figure 13 shows such runtime overhead with and without dynamic analysis. To measure such overheads, we collected the performance data with original approach and compare them with those with only the online analysis but without applying the constrained-equivalent rules. As Figure 13 shows, the dynamic analysis will introduce very little overhead, which is less than 1% on average. The low overhead is due to two main reasons. First, the dynamic analysis typically only needs to check a few registers and condition codes. And the percentage of the rules that requires dynamic analysis is not very high. Second, the relaxed register and condition-code operands are usually updated very quickly, so only a very small number of instructions need to be analyzed in practice. Both factors greatly reduce the analysis overhead.

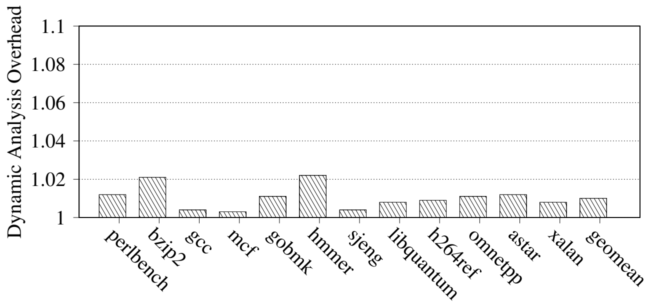


Figure 13: The performance overhead of dynamic analysis.

6 Related Work

To improve the efficiency of the translated host binaries, many manual optimization techniques have been proposed. For example, some try to efficiently translate guest single-instruction-multiple-data (SIMD) instructions [7, 10, 16]. Another work proposes to leverage host hardware features to apply post-optimization to host binary code after the translation [30]. Some recent work also proposes to optimize dynamically-generated guest binary code [8]. Different from those approaches, most of which rely on manually constructed translation rules, our enhanced learning-based approach proposed in this paper can automatically learn binary translation rules.

Previous work in [1] also proposes to use peephole superoptimizer to generate binary translation rules for static binary translators. For each potential guest code sequence, an exhaustive search is employed to explore all possible sequences of host instructions to examine their equivalence. However, it takes a very long time to collect sufficient translation rules, i.e., could be up to one week as mentioned in the paper. Moreover, due to the exponential increase in the number of possible instruction sequences, this approach can only generate translation rules with a guest code sequence of up to 3 instructions. This can significantly limit the quality of the generated translation rules because many high-quality translation rules have more than 3 guest instructions.

Although the learning-based approach was originally proposed in [23], our enhanced learning-based approach differs from the original approach in a significant way. Our proposed enhanced approach allows relaxation of semantic equivalence, thus can learn constrained-equivalent translation rules while the original approach simply discards them. These relaxed translation rules can improve the total coverage of the guest binaries and improve the yield of rule generation. More importantly, these constrained-equivalent translation rules can further improve the performance of the translated host binary code.

Another DBT system, HQEMU [9], which is also based on QEMU, translates guest binary code into LLVM intermediate representation (IR) and then leverages LLVM JIT to generate more optimized binary code. However, the overhead introduced by the LLVM optimization can offset the benefit gained from the optimized host binary code, especially for short-running guest binaries. Moreover, due to the lack of source-level information in the LLVM IR translated from the guest binary code, e.g., type information, it is quite challenging to take full advantage of the LLVM optimization. In contrast, the translation overhead for applying the learned translation rules are much smaller, and no additional information is required to apply the learned rules.

There has been a lot of research to improve the performance of the DBT system itself [3–5, 11, 12, 24, 25]. These methods can typically be used in conjunction with our approach to further improve their performance.

7 Conclusion

As one of the core enabling technologies, DBT has been extensively used in many important applications. To improve the efficiency of DBT systems, this paper proposes an enhanced learning-based approach, which can automatically learn optimized binary translation rules. The learned translation rules can then be applied to a DBT system to generate more efficient host binary code. Compared to the original learning approach, our enhanced learning-based approach relaxes the semantic equivalence requirements to allow more efficient constrained-equivalent translation rules. We redesign the original learning process and the verification engine to accommodate such constrained equivalence. Moreover, to preserve the correct semantics of the translated code when such constrained-equivalent translation rules are applied, a lightweight online analysis is employed in the enhanced DBT system to check the constraining conditions. The constrained-equivalent translation rules are applied only when the constraining conditions are satisfied. We have implemented the proposed approach in a prototype and extended a widely-used DBT system, i.e., QEMU, to accept such enhanced translation rules through learning.

Experimental results on SPEC CINT2006 show that the proposed approach can improve the dynamic coverage of the translation from 55.7% to 69.1% and the static coverage from 52.2% to 61.8%, compared to the original approach. Moreover, up to 1.65X performance speedup with an average of 1.19X are observed.

Acknowledgments

We are very grateful to our shepherd, Edouard Bugnion, and the anonymous reviewers for their valuable feedback and comments. This work is supported in part by the National Natural Science Foundation of China (No. 61672160), Shanghai Municipal Science and Technology Major Project (No.2018SHZDZX01) and ZJLab, Shanghai Science and Technology Development Funds (17511102200) and the National Science Foundation under the grant number CNS-1514444.

References

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

[2] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATC '05, pages 41–46, Berkeley, CA, USA, 2005. USENIX Association.

[3] Chao-Jui Chang, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. 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, pages 117–128, New York, NY, USA, 2014. ACM.

[4] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. Cross-isa machine emulation for multicores. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 210–220, Piscataway, NJ, USA, 2017. IEEE Press.

[5] Amanieu D'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. Low overhead dynamic binary translation on arm. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 333–346, New York, NY, USA, 2017. ACM.

[6] Peter Feiner, Angela Demke Brown, and Ashvin Goel. Comprehensive Kernel Instrumentation via Dynamic Binary Translation. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 135–146, New York, NY, USA, 2012. ACM.

[7] Sheng-Yu Fu, Ding-Yong Hong, Yu-Ping Liu, Jan-Jan Wu, and Wei-Chung Hsu. Dynamic translation of structured loads/stores and register mapping for architectures with simd extensions. In *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2017, pages 31–40, New York, NY, USA, 2017. ACM.

[8] Byron Hawkins, Brian Demsky, Derek Bruening, and Qin Zhao. Optimizing binary translation of dynamically generated code. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 68–78, Washington, DC, USA, 2015. IEEE Computer Society.

[9] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. Hqemu: a multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 104–113. ACM, 2012.

[10] Ding-Yong Hong, Yu-Ping Liu, Sheng-Yu Fu, Jan-Jan Wu, and Wei-Chung Hsu. Improving simd parallelism via dynamic binary translation. *ACM Trans. Embed. Comput. Syst.*, 17(3):61:1–61:27, February 2018.

[11] Chun-Chen Hsu, Pangfeng Liu, Jan-Jan Wu, Pen-Chung Yew, Ding-Yong Hong, Wei-Chung Hsu, and Chien-Min Wang. 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, pages 23–32, New York, NY, USA, 2013. ACM.

[12] Ning Jia, Chun Yang, Jing Wang, Dong Tong, and Keyi Wang. 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, pages 1–12, New York, NY, USA, 2013. ACM.

[13] Piyus Kedia and Sorav Bansal. Fast Dynamic Binary Translation for the Kernel. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 101–115, New York, NY, USA, 2013. ACM.

[14] Dohyeong Kim, William N. Sumner, Xiangyu Zhang, Dongyan Xu, and Hira Agrawal. Reuse-oriented Reverse Engineering of Functional Components from x86 Binaries. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1128–1139, New York, NY, USA, 2014. ACM.

[15] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[16] Jianhui Li, Qi Zhang, Shu Xu, and Bo Huang. Optimizing dynamic binary translation for simd instructions. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 269–280, Washington, DC, USA, 2006. IEEE Computer Society.

[17] Juanru Li, Zhiqiang Lin, Juan Caballero, Yuanyuan Zhang, and Dawu Gu. K-Hunt: Pinpointing Insecure Cryptographic Keys from Execution Traces. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 412–425, New York, NY, USA, 2018. ACM.

[18] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 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, pages 190–200, New York, NY, USA, 2005. ACM.

[19] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 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, pages 337–348, New York, NY, USA, 2012. ACM.

[20] Aashish Mittal, Dushyant Bansal, Sorav Bansal, and Varun Sethi. Efficient Virtualization on Embedded Power Architecture®Platforms. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 445–458, New York, NY, USA, 2013. ACM.

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

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

[23] Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. Enhancing Cross-ISA DBT Through Automatically Learned Translation Rules. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 84–97, New York, NY, USA, 2018. ACM.

[24] Wenwen Wang, Jiacheng Wu, Xiaoli Gong, Tao Li, and Pen-Chung Yew. Improving dynamically-generated code performance on dynamic binary translators. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '18, pages 17–30, New York, NY, USA, 2018. ACM.

[25] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. A General Persistent Code Caching Framework for Dynamic Binary Translation (DBT). In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 591–603, Berkeley, CA, USA, 2016. USENIX Association.

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

[27] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. Coremu: A scalable and portable parallel full-system emulator. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 213–222, New York, NY, USA, 2011. ACM.

[28] Qifan Yang, Zhenhua Li, Yunhao Liu, Hai Long, Yuanchao Huang, Jiaming He, Tianyin Xu, and Ennan Zhai. Mobile Gaming on Personal Computers with Direct Android Emulation. In *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking*, MobiCom '19, New York, NY, USA, 2019. ACM.

[29] W. Zhang, X. Ji, Y. Lu, H. Wang, H. Chen, and P. Yew. Prophet: A parallel instruction-oriented many-core simulator. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2939–2952, Oct 2017.

[30] Xiaochun Zhang, Qi Guo, Yunji Chen, Tianshi Chen, and Weiwu Hu. 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, pages 246–256, Washington, DC, USA, 2015. IEEE Computer Society.