More with Less – Deriving More Translation Rules with Less Training Data for DBTs Using Parameterization

Jinhu Jiang[†], Rongchao Dong[†], Zhongjun Zhou[†],
Changheng Song[†], Wenwen Wang[+], Pen-Chung Yew[§], Weihua Zhang[†♯*‡]
Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University[†]
Shanghai Institute of Intelligent Electronics & Systems[‡]
Big Data Institute, Fudan University[♯]
Human Phenome Institute, Fudan University[*]
Department of Computer Science, University of Georgia[+]
Department of Computer Science and Engineering, University of Minnesota, Twin Cities[§]
{jiangjinhu, 16302010007, 19212010010, 17212010032, zhangweihua}@fudan.edu.cn,
wenwen@cs.uga.edu, yew@umn.edu

***Abstract*— Dynamic binary translation (DBT) is widely used in system virtualization and many other important applications. To achieve a higher translation quality, a *learning-based* approach has been recently proposed to automatically *learn* semantically-equivalent translation rules. Because translation rules directly impact the quality and performance of the translated host codes, one of the key issues is to collect as many translation rules as possible through minimal training data set. The collected translation rules should also cover (i.e. apply to) as many guest binary instructions or code sequences as possible at the runtime. For those guest binary instructions that are not covered by the learned rules, emulation has to be used, which will incur additional runtime overhead. Prior learning-based DBT systems only achieve an average of about 69% dynamic code coverage for SPEC CINT 2006.**

In this paper, we propose a novel parameterization approach to take advantage of the regularity and the well-structured format in most modern ISAs. It allows us to extend the learned translation rules to include instructions or instruction sequences of similar structures or characteristics that are not covered in the training set. More translation rules can thus be harvested from the same training set. Experimental results on QEMU 4.1 show that using such a parameterization approach we can expand the learned 2,724 rules to 86,423 applicable rules for SPEC CINT 2006. Its code coverage can also be expanded from about 69.7% to about 95.5% with a 24% performance improvement compared to enhanced learning-based approach.

I. Introduction

Dynamic binary translation (DBT) is a key enabling technology that has been used in many important applications, such as system virtualization and emulation [14], [24], architecture simulation [15], [23], [25], program analysis [4], [11], and mobile computation offloading [22].

In essence, a DBT system translates binary code from a *guest* instruction set architecture (ISA) to a *host* ISA, which can be different from or the same as the guest ISA. The translation process is governed by the *translation rules* (or *rules* for short). Each of those rules maps one or more guest instructions into *semantically-equivalent* host instructions. By executing the translated host binary, the DBT system can emulate the guest binary on a host machine. In this way, we can achieve a better power efficiency and/or performance by offloading applications from mobile devices to more powerful servers, or allow code migration to facilitate workload consolidation and better resource provisioning in data centers.

Recently, a *learning-based* approach was proposed to mitigate the engineering effort required to develop a DBT system, as well as to improve the quality of the translated binary code [18]. This approach automatically learns translation rules from the abundance of source codes. The main idea is that the binaries (in *different* ISAs) compiled from the *same* source program should be semantically equivalent. For example, the X86 binary and the ARM binary compiled from the same C source code should be semantically equivalent. By narrowing the scope to program statements, the binary code sequences (in different ISAs) compiled from the same *source statement* should also be semantically equivalent. The mapping of these binary instructions or sequences from the host to guest binaries form the basis of the *learned* translation rules. A formal verification step, based on symbolic execution methods [13], is employed to verify and guarantee the the semantic equivalence of the learned rules. Their experimental results show that such an approach can produce higher quality translated host binaries and ease the engineering effort of manually producing translation rules in DBTs.

However, some recent enhancement [16] shows that this learning-based approach still cannot achieve an expected high code *coverage* at runtime. It only improves from an average coverage of 55.7% [18] to about 69.7% [16] for SPEC CINT 2006. The *coverage* here means the percentage of guest binary code that can be translated using the learned translation rules

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00043

at runtime. In general, a higher coverage means more guest binary code can be translated by the learned rules at runtime, instead of using more time-consuming emulation. It can thus achieve a higher performance.

The challenge here is that, unless a sufficiently large collection of training programs is used, it will be difficult to produce a comprehensive set of learned translation rules to improve the overall code coverage. Furthermore, it is well known that only a small percentage of the code is executed during the runtime. Our experimental data on SPEC CINT 2006 shows that less than 5% of the statements are executed at runtime (Section II). Hence, it will require a much larger training data set to harvest a comprehensive set of translation rules. Obviously, the larger the training data set is, the more time consuming the training process will become.

To overcome these challenges, we proposed a parameterization approach that takes advantage of the well-structured format of the modern ISAs, and extend the learned translation rules to derive "similarly-formatted" rules. For example, the parameterization can be applied in two dimensions that include *opcode* and *addressing mode* (Section IV). Such an approach comes naturally from the fact that most modern ISAs are designed with these two major components. Parameterizing opcode and addressing mode allows the learned rules to be extended to instructions of the same opcode types or addressing modes that are not in the training set, and be included in the translation rules. For example, a learned translation rule with an "*add*" instruction of certain addressing mode can be "parameterized" or "generalized" to produce the rule for the "*subtract*" instruction with the same addressing mode but is not in the training set, i.e. not "learned" yet.

We find some constraints are needed when we generalize along these dimensions. These constraints are needed to guarantee the correctness when these *derived* translation rules are applied. For example, two instructions with the opcodes of the same type (e.g. add and subtract) that have the same addressing mode, but operands of one instruction type might be *mutable* (e.g. add) while the other might be *immutable* (e.g. subtract). More details are in Section IV-C.

Using the above approach, we have implemented a prototype on a popular retargetable DBT system QEMU 4.1 to *automatically* extend learned rules to parameterized rules. It involves about 800 lines of code. Experimental studies show that the proposed parameterization approach is quite effective. It can greatly expand the learned rules from the same training data set. Through parameterization, we can derive 86,423 rules from the 2,724 initially learned rules for SPEC CINT 2006. Only a few instructions (7 instructions in this case) cannot be derived through such parameterization.

With the derived set of the translation rules from the original training data set, the dynamic coverage can be enhanced from 69.7% to 95.5%. Moreover, it can achieve about 24% performance improvement compared to that of the state-of-the-art learning-based approach [16].

In summary, this paper makes the following contributions:

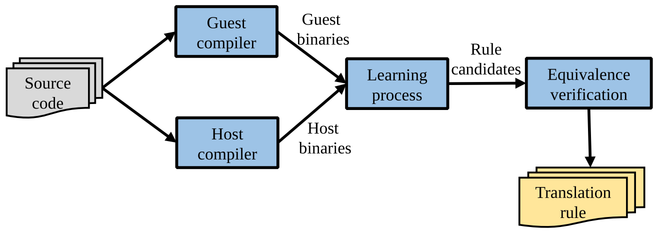

Fig. 1. The workflow of a learning-based DBT.

- We propose a parameterization approach for the learning-based DBT to derive more translation rules from the same training data set. It can reduce the training time and improve dynamic coverage at runtime.
- We implement the proposed learning-based approach in a prototype, which includes a parameterization framework and a DBT system based on QEMU to accept the parameterized translation rules[1].
- We conduct extensive empirical studies to evaluate the proposed parameterization approach. Experimental results show that our approach can improve the coverage to 95.5%, and achieve an average of 1.24X speedup compared to a state-of-the-art learning-based approach.

The rest of the paper is organized as follows. In Section II, we present the motivation and some challenges in increasing the translation rules from learning. Section III gives an overview of our parameterization approach. In Section IV, we discuss the design of our parameterization approach and how they are applied. In Section V, we present some implementation details and evaluate the proposed approach with some experimental results. Section VI presents some related work, and Section VII concludes the paper.

II. Background

In this sections, we give some background of learning-based DBT systems and identify some challenges in them.

A. Learning-Based DBT

Most existing DBT systems are implemented like a typical compiler. Guest instructions are first translated into an intermediate representation (IR), in which each guest instruction is translated into one or several pseudo instructions in IR. These pseudo instructions are then translated into the target host binaries. All of the translation rules used are provided manually, which requires significant engineering effort. Some compiler optimizations may be applied during the process to produce better host binary code.

Learning-based DBT systems [16], [18] are done differently. The major difference is in how the translation rules are generated. Instead of manually generating rules, learning-based DBT systems automatically learn translation rules from the source code. Figure 1 shows the workflow. The same compiler of guest and host (usually *gcc* or *LLVM*) will generate both

[1]The parameterized learning-based DBT has been released as an open-source project at https://github.com/FudanPPI/Rule_based_DBT

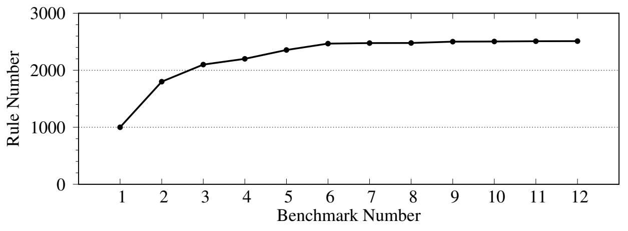

Fig. 2. The number of rules learned with an increasing number of training benchmarks using SPEC CINT 2006.

guest binary and host binary from the same source code. The binary instructions or instruction sequence in the guest and host binaries from the *same source statement* are extracted as the rule candidate. In the verification step, using symbolic execution techniques, if they are proven to be semantically equivalent, a translation rule can be generated. This learning process can be done automatically with minimal manual intervention. These training binaries can be generated from an optimizing compiler. Hence, such a method can produce high-quality rules with less engineering effort.

Another advantage of learning-based approach is that it skips the step of going through IR, i.e. it directly translates guest to host binary. It avoids the problem of "code expansion" through IR, i.e. in most cases, *multiple* IR pseudo-instructions in IR are needed to emulate *one* guest instruction, and *multiple* host instructions may also be needed to emulate *one* IR pseudo-instruction. It thus avoids such a "multiplying" effect that requires many more host instructions to emulate the guest binary than required [18].

B. Challenges in Learning-Based DBT

As mentioned earlier, the main challenges in learning-based approaches are their low coverage and the requirement of a large training set. One possible way to improve the coverage is to increase the size of the training set to obtain more rules. Intuitively, if we keep increasing the size of the training set and collect more rules along the way, we should be able to harvest enough rules that can achieve high coverage at runtime. However, it does not work well in practice.

To better understand this challenge, we measure the increase of the learned rules and their coverage using SPEC CINT 2006 as an example. In this study, one additional benchmark in SPEC CINT 2006 is randomly selected and added to the training set. The number of harvested learned rules collected after each addition of one randomly selected benchmark are shown in Figure 2. [2] As the results show, the number of learned rules increases very slowly after about 6 benchmarks in the training set. Moreover, as shown on the dotted line (w/o para) in Figure 16, the dynamic coverage of the enhanced learning-based approach [16] also becomes stagnant after a training set of 6 benchmarks and plateaued at around 69.7% on average.

[2]The measurement is repeated multiple times each time with a different randomly selected benchmark. As the results are quite consistent, we only show the result of one randomly selected benchmark (In this case, *perlbench* is the first chosen benchmark), which is enough to illustrate the problem.

TABLE I
THE STATISTICS ON THE NUMBER OF RULES FROM SPEC CINT 2006 USING THE ENHANCED LEARNING APPROACH.

Benchmark	Statement	Candidate	Learned rule	Unique rule
perlbench	48634	31496	13171	968
bzip2	3096	2165	811	140
gcc	143190	77659	36054	1405
mcf	531	372	185	39
gobmk	27975	15264	3384	351
hmmer	10213	4980	1871	210
sjeng	4933	2976	734	136
libquantum	1012	342	127	42
h264ref	20165	12960	3264	409
omnetpp	14067	3852	2125	169
astar	1516	812	291	66
xalancbmk	71040	33386	16161	397
Avg.	28864	15522	6514	361
Percent%	100%	53.8%	22.6%	1.3%

The reasons behind such results are two folds. First, in the training phase, the rules that can be learned from a training set depend on the composition of the applications in the training set. Some instructions and instruction sequences occur more frequently than others. For example, *add*, *eor* and *rsc* are arithmetic and logical operation instructions. There are about 1178 rules related to the *add* instruction that can be learned in SPEC CINT 2006. However, only about 34 rules are related to *eor*, and no rule for *rsc*. It is similar for different addressing modes. Predicting which application should be included in the training set to get the translation rules we need is extremely difficult. Simply adding more applications into the training set will be very time-consuming and ineffective. Second, during the rule application, it is also quite unpredictable what translation rules will be needed. Therefore, improving dynamic coverage is very difficult unless the learned translation rules cover the majority of the instructions and addressing modes in the guest ISA, which may require a huge training set and will be quite time-consuming.

Table. I shows some results on the rules learned by the enhanced learning-based approach [16] using SPEC CINT 2006. In the table, *"Statements"* shows the number of source statements in an application. As mentioned earlier, the guest binary instructions and the host binary instructions generated from the *same* statements by a compiler are collected as rule candidates [16], [18]. In the table, *"Candidates"* shows the number of such rule candidates collected from the source statements. *"Learned rules"* shows the number of rule candidates that pass the verification process and can be included in the translation rules. *"Unique rules"* shows the final size of the learned translation rules after eliminating duplication.

As the results show, the learning process is quite inefficient. There are three main reasons for this. Firstly, the mapping between the binary instructions and their corresponding source statements is established by the debugging tool such as GDB. The mapping may be inaccurate because compiler optimization can cause binaries from multiple statements to be merged, eliminated or scattered. It can also cause the binaries to be mistakenly mapped to the wrong statements, or lose the connection. Thus, only around 53.8% of the statements can

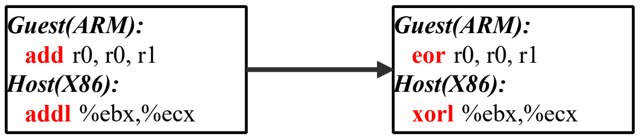

Fig. 3. An example of generalizing the opcode.

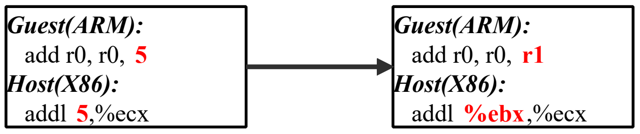

Fig. 4. An example of generalizing addressing mode.

produce rule candidates.

Secondly, the verification process using symbolic execution techniques is very strict. For example, guest register/memory operands must have their corresponding host register/memory operands of the same type. In addition, these mappings must be one-to-one, i.e. the number of operands must be the same. But, in different ISAs, the number for operands and the types of the operands could be different. In such cases, the strict symbolic execution techniques will consider the test of semantic equivalence fails. Moreover, there exits inconsistency between different ISAs. For example, they may have different ABIs, different condition flags, and different architecture-specific instructions, which also impedes the verification process. Therefore, only around 22.6% of statements can produce translation rules.

Finally, the rules generated from different statements may be the same. So, a merging process is needed to remove the duplication. With all those considerations, the final number of translation rules we can get is only around 1.3% of the source statements on average. The average coverage is only around 69.7% for SPEC CINT 2006 [16].

III. Parameterization Framework

As mentioned earlier, most instructions have two major components, namely opcode and addressing mode. There can be condition flags for arithmetic and logic instructions as side effects. Figure 3 shows an example with the ARM ISA as the guest and X86 ISA as the host. In the ARM ISA, *add* and *eor* have the same addressing mode. Assume we have learned a translation rule for the *add* instruction after the verification process, as shown in the left box. We can derive a similar translation rule for the *eor* instruction with the same addressing mode as shown on the right, even if it is not included in the training set. Thus, parameterizing (or generalizing) the opcodes in the learned rules can greatly increase the number of translation rules without increasing the training set. In our study, after generalizing the 1178 learned rules involving the *add* instruction to the *eor* instruction, we can get 1178 translation rules that involve the *eor* instruction without increasing the training set.

Such a parameterization/generalization process also works for the addressing modes. Figure 4 shows such an example. In this example, we replace the addressing mode of *immediate* in the rule (shown in the left box of the figure) with the register addressing mode, we can get a new rule shown on the right, which is also valid. Notice that such a parameterization process can be applied to translation rules that involve multiple-instruction code sequences as well, which can increase the set of the translation rules even further.

Based on the above observations, we proposed our parameterization approach for the learning-based DBT systems. Figure 5 shows its workflow. In the workflow, we first classify all instructions in the ISA into several subsets based on their opcodes and addressing modes.To guarantee the validity of the parameterized translation rules, the same verification process is used to check their correctness. Moreover, the constraining conditions (if exist) are also provided to further guarantee the semantic equivalence when the translation rules are applied at runtime. More details are in the next section.

IV. Parameterization Workflow

Rule parameterization contains four steps: *classification, parameterization, verification*, and *merging*. The classification step classifies instructions into different subsets, and only the instructions in the same subset share the same parameterization rule. Then, the learned rules are parameterized along the two dimensions of opcode and addressing mode. A verification process is used to ensure the semantic equivalence. For some non-equivalent rules, some additional constraints might help to guarantee the correctness when applied at runtime. At last, the redundant rules are removed in the merging stage.

A. Classification

Most ISAs embed their data types such as *integer*, *float point*, and *vector* in their opcodes, e.g., integer add or floating-point add. These instructions of different data types are executed on different functional units with different data formats. Hence, the translation rules for instructions of one data type usually cannot be parameterized to derive the instructions of different data types. We thus classify instructions first into different subsets according to their data types.

For instructions of the same data type, we further classify/divide them according to two guidelines. The first is that the instructions in the same subgroup must have the same encoding format, e.g. they should have the same length for X86 or the same R-type for MIPS. The second guideline is that the instructions in the same subgroup must perform similar types of operations, e.g. arithmetic and logic, data transfer, or branch. We use these two guidelines because they are easy to implement. Moreover, the instructions in the subgroups formed by these two guidelines tend to have very similar characteristics and are easier to parameterize.

Using the ARM ISA as an example, we first classify all instructions according to data types, such as integer and float point. Then, for integer instructions, we further classify them

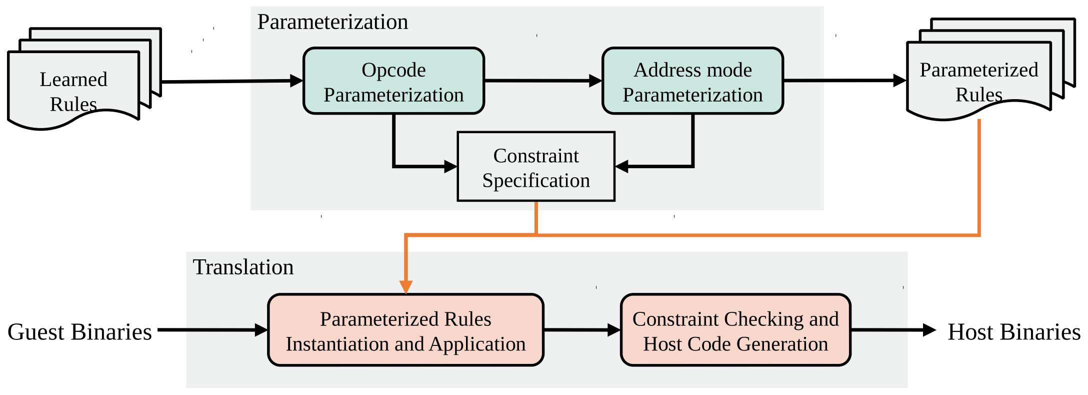

Fig. 5. The workflow of rule parameterization.

into five subsets according to the two guidelines mentioned above. Those five subsets are (1) arithmetic and logic, such as *add*, *and* and *sub*; (2) Data transfer from memory to registers, such as *mov* and *ldr*; (3). Data transfer from registers to memory, such as *str*; (4) Compare, such as *cmp* and *tst*; (5) Remaining that cannot be classified into the above subgroups, such as *b*, *push* and *pop*. Note that the instructions in each subgroup use the same parameterized rules, and each translation rule involves a guest part and a host part. Therefore, if a learned rule can be parameterized (i.e. is ”parameterizable”), both the guest part and the host part should also be parameterizable, i.e. each should belong to a subgroup in its corresponding ISA. If a guest instruction belongs to a specific subgroup, its translated host instruction should also fall into its corresponding host instruction subgroup. As most of the ISA designs follow similar design principles in their formats, we find ARM and Intel X86 in our prototype are no exception and can be divided into similar subgroups. After all of the instructions in the ISA are classified into a handful of different subgroups, a pseudo opcode is assigned to each subgroup. For simplicity, we denote the *ith* instruction subgroup of the guest ISA as $guestpara\_op_i$, and its corresponding host instruction subgroup as $hostpara\_op_i$.

B. Parameterization

After the instructions are classified into subgroups, a parameterization process is used to derive more translation rules from the learned rules. As mentioned earlier, our parameterization is done along two dimensions, namely, opcodes and addressing modes. Here, we again use the translation rules from a RISC ISA such as ARM to a CISC ISA such as X86 as an example.

To parameterize the opcodes of the instructions in a subgroup, we replace their opcodes by a pseudo opcode ($para\_op_i$) assigned to the subgroup. Their addressing modes can be *register*, *memory* or *immediate*. However, there can be only one addressing mode for each operand in a rule. Parameterizing the addressing mode is to generalize operands to all possible addressing modes of this subgroup of instructions without learning them all from the training set. Several simple guidelines are used during the generalization. The first is that the target operand (i.e. the operand that store the result of the operation) cannot be an *immediate*. The second is that the operands of non-load/store instructions cannot be parameterized to *memory* in a RISC ISA. The third is that the source operand of a *load* operation and the target operand of a *store* operation must be in the *memory* mode.

Condition flags are common in many ISAs, such as eflags in X86 and CPSR in ARM. When a rule is learned, its related condition flags are included as a part of the rule even if they are implicit in the instructions, i.e. they are considered as the ”side effects” of the execution. In a subgroup, some instructions will set condition flags while the others do not. For example, both *ands* and *and* instructions are in the arithmetic and logic subgroup of ARM. However, *ands* will set condition flags, but *and* does not. We thus cannot use the same pseudo opcode for them unless we further divide them into different subsets and assign different pseudo opcodes.

To avoid too many of such subsets, we can emulate the condition flags by mapping each flag in a memory location, and update them after the instruction execution that has such side effects. But doing so will involve a lot of memory overhead. We thus use host condition flags to emulate their corresponding guest condition flags as much as possible unless they are not available. We call this process *condition flags delegation*. More details are in Section IV-D.

A cautionary note here is that, in a learned rule with a sequence of instructions, some auxiliary instructions may exist that are unrelated to the emulation of the guest instructions. For example, some auxiliary instructions may be needed due to register spilling and other considerations on the host. These instructions may not have corresponding instructions in the guest binary. When a rule is parameterized, these auxiliary instructions should be excluded from the parameterization.

Figure 6 shows a parameterization example. The top left of the figure is the original rule used to translate an *add* instruction, and there is an auxiliary instruction *movl* for register spilling. After the opcode and the addressing modes are parameterized, we got a parameterized rule as shown in the top right of the figure. Note that *movl* instruction is not

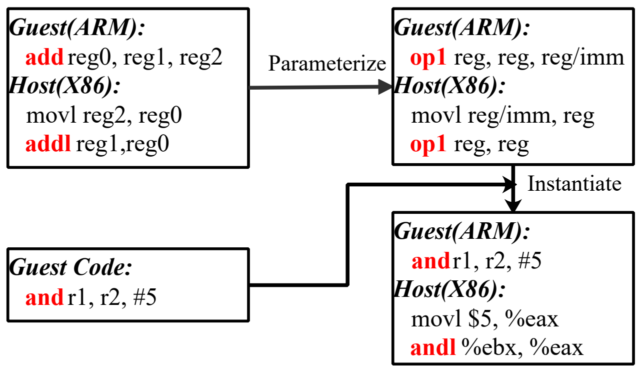

Fig. 6. A rule-parameterization example.

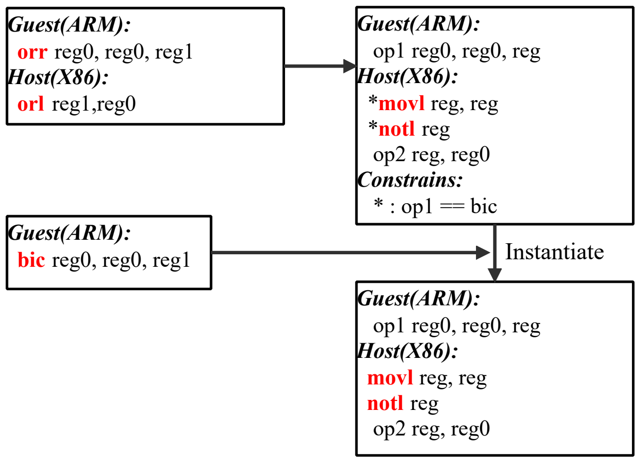

Fig. 7. An example of extending a rule from a simple instruction *orr* to a complex instruction *bic*.

parameterized. When we use it to derive an *and* instruction shown at the bottom left of the figure, this rule is instantiated and its opcode is substituted with the opcode *and* with its operands in their corresponding addressing modes. The auxiliary instruction is unchanged.

C. Verification and Constraints

After the parameterization step, we can derive many new translation rules from the parameterization. But these derived rules may not be semantic equivalent. So a verification process is needed to confirm their semantic equivalence. We use the same symbolic execution technique as in the original learning-based approach [16], [18]. In the verification process, we first instantiate all possible derived rules from the parameterized rule, and verify each derived rule to check its semantic equivalence. We can harvest the derived rules that pass the verification process. For those that do not pass the verification process, some constraints may be added when they are applied at runtime and make them constrained semantic equivalence. In the following subsections, we describe how to add the constraints in those cases.

1) Opcode Constraints: Although the instructions in the same subgroup have similar addressing modes for their operands, their property may be different. As the example mentioned earlier, the two operands in an *add* instruction are commutative, while those in a *sub* instruction are non-commutative. If the non-commutative property is not observed, the result will be wrong. One way to safeguard this property is to divide the instructions in this subgroup into different subsets as in the case of different side effects from the condition flags. However, it could lose some of the possible derivable rules from parameterization. Thus, we first assume these operands are commutative. After the parameterization, if an instruction has non-commutative operands, the verification process will drop them.

Another problem with the opcode parameterization is that two guest instructions in the subgroup may have the same instruction format and execute the same operation, but one of them have some additional features and semantics. For example, In ARM ISA, both *"orr reg0, reg0, reg1"* and *"bic reg0, reg0, reg1"* have the same instruction format and perform the same *or* operation. While *orr* uses the value in reg1 as its operand, *bic* will first *invert* the value before it is used. There exist other such instruction pairs, such as, *mvn* and *mov*.

Between the instructions in the pair, the semantics of one instruction is more complex than the other. After the rule of the instruction with simpler semantics, such as *orr*, is parameterized, it cannot be applied directly to the instruction with more complex semantics. To make the parameterization more applicable in such cases, some auxiliary host instructions may be added to emulate the uncovered semantics of the complex instruction. Alternatively, some constraints associated with the complex instruction can be included to make it constrained semantically equivalent as in [16]. In some cases, we can have both in the parameterization.

Figure 7 shows an example for *orr* and *bic*. In this example, we have learned a rule for *orr*. To parameterize it but still cover *bic*, two auxiliary instructions *"movl reg, reg"* and *"notl reg"* are added after the parameterization, and *bic* is noted in the constraint condition. When *bic* is translated by this rule, the auxiliary instructions are added to the host codes.

2) Addressing Mode Constraints: In a learned rule, there may exist data dependency among the operands of instructions. These dependency relations may exist within an instruction or among instructions in the rule. When these instructions are parameterized, these dependency relations should be preserved to guarantee the correctness. Before a parameterized rule is applied, the data dependency in translated guest codes should be checked. Only when the dependency relations in the guest code are consistent with those in the rule, the rule can be applied.

Figure 8 shows such an example. In Figure 8(a), *reg0* is both a source operand and the result operand. Thus, there is a dependence between its source and its result. Although the parameterized instruction in Figure 8(b) is also an *add* instruction, there is no dependency among its operands. To

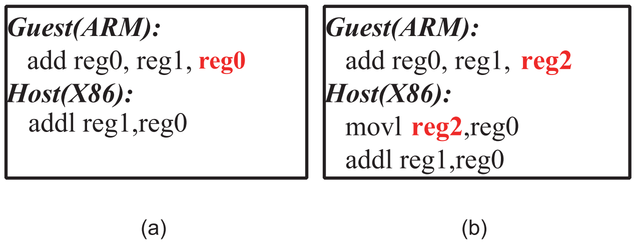

Fig. 8. An example of a parameterized rule with a data dependence.

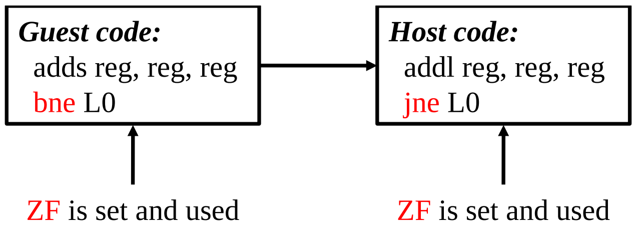

Fig. 10. An example that uses host condition flag to optimize the translation.

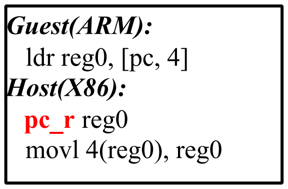

Fig. 9. Different semantics between ARM binaries and X86 binaries.

preserve the dependency relation, an auxiliary instruction *"movl reg2, reg0"* is added to the parameterized rule as shown in the figure.

Figure **??** is another example, which can be used to illustrate the dependency relation among different instructions. There is a register dependency on *reg0* among three guest instructions, and there is also a memory dependency between *ldr* and *str*. If we want to parameterize this translation rule, these dependency relations must be preserved and enforced after the parameterization besides just substituting the opcodes and the operands with different addressing modes.

Different ISAs may use registers in different ways. For example, *PC* register (i.e. the program counter) can sometimes be used as a general-purpose register in ARM ISA. However, it cannot be used in such a manner in X86 ISA. When addressing modes are parameterized, some constraints should be included to deal with such differences. Figure 9 is such a case, where *PC* register is used like a general-purpose register to calculate a PC-relative address in the ARM guest code. When a parameterized rule is instantiated for this instruction, the X86 host instruction cannot use *PC* register, but another general-purpose register instead.

D. Rule Application

After the parameterization, there may be some duplicated rules. For example, *"add r0, r0, r1"* and *"sub r0, r1, 5"* will both be parameterized as *"op1 reg, reg, reg/imm"*. So, a merging process is necessary to remove the redundant rules. For each parameterized rule, it will be compared with other existing rules to check whether they are the same. If so, it will be removed.

After the rules are parameterized and verified, they can be included in the translation rules, and used for translation. When a guest instruction is being translated, it is first parameterized to retrieve the rules for translation. If a rule is matched, it is instantiated with the opcode and the operands of the *guest* instruction, and used for translation. All constraints associated with the translation rule will also be checked, and the necessary auxiliary instructions will be added to guarantee the correctness of translation. Compared to the scheme in [18], only two additional simple steps are needed that include (1) guest instruction parameterization and (2) matched rule instantiation. Both incur very little additional overhead as the experimental results in Section V-B1 show.

As mentioned in Section IV-B, condition flags are not included in the parameterized rules but are handled in the translated host code using *condition flags delegation*. It is because a large part of the condition codes are the same in all ISAs. For example, both X86 and ARM have the zero flag (ZF), overflow flag (OF), etc. We can thus emulate these guest condition flags using their corresponding host condition flags. When a guest instruction is being translated, we will first check whether the guest instructions and the corresponding host instructions in the rule will set the same condition flags or not. If so, the host condition flags are used for emulation. Otherwise, we will further check whether the host condition flags will be modified (or "killed") by the following host instructions before the flags are used.

Such a check may go down a long distance, or we may not be able to confirm whether the flags are still needed or not. However, in most cases, the compiler will generate code in such a way that the flags are used in the next couple of instructions, if needed. Therefore, in our current implementation, we only check *three* instructions following a condition flag-setting instruction. If the flags are used within that window without other modification, they are emulated by the corresponding host flags. Otherwise, if the flags are not "killed" within that window, they are emulated by their corresponding memory locations to guarantee the correctness. Such an approach can reduce a large number of unnecessary memory operations to emulate condition flags.

Figure 10 shows such a example. Both the guest *adds* instruction and the host *addl* instruction set the ZF (zero) flag. The instructions *bne* and *jne* will branch according to the value of the ZF flag. In this case, the host ZF flag is not modified by the other instructions after it is set by *addl*. Hence, no memory operation is needed.

V. Evaluation

We have implemented a prototype to evaluate the proposed scheme and address the following research issues in this section: (1) How much performance gain and code coverage improvement can the parameterization approach achieve compared to the existing learning-based ones? (2) What are the factors that contribute to their improvement? (3) How much does each factor contribute to such improvement? (4) How does our approach impact the translation rules learned?

A. Experimental Setup

To evaluate the proposed parameterized learning-based DBT system, we have implemented a prototype based on QEMU 4.1, which was the latest released version at that time and was also the platform used in prior learning-based approaches. QEMU supports both the user mode and the system mode. Currently, we have only implemented our approach in the user mode, as in [16] and [18], for comparison. Supporting the system mode requires further consideration, which is beyond the scope of this paper.

The implementation of parameterization takes around 800 lines of code. About 500 lines of code are used to extend a learned rule to a parameterized rule during the rule generation phase. About 300 lines of code are used to support the application of the parameterized rules at runtime. A guest instruction is parameterized first, and a hash algorithm is used to retrieve the translation rules from a hash table. The matched rule will then be instantiated to generate host instructions as described in Section IV-D.

We use ARM-v8 as our guest ISA and Intel X86 as our host ISA. The implemented prototype was run on a 3.33GHz Intel Core i7-980x with six cores. It has 16GB of RAM, and the OS is 32-bit Ubuntu 14.04 with Linux 3.13.

The guest binary codes (for learning and translation) and the host binary codes (for learning and for comparison) are generated by the LLVM compiler (version 3.8.0) with the same -O2 optimization. The training process, which is to learn and produce the translation rules, is the same as that in the previous learning-based DBT [16], [18]. It is described in detail in Section II-A. The SPEC CINT 2006 benchmark suite with 12 benchmarks is also used in our study for performance comparison. As in [16], [18], the rules learned from the other 11 benchmarks are applied to the 12th benchmark. This process is repeated on each individual benchmark. The *reference input* is used, which has the longest-running time. This gives us a better understanding on the performance impact of the *code coverage* and the *quality* of the translated code with minimal influence from the translation overhead. We run the 12 benchmarks ten times and their average is used.

B. Performance Improvement

We compare the performance of our approach, QEMU 4.1, and prior learning-based DBT systems [16], [18]. We then analyze how different design choices in our approach impact the overall performance.

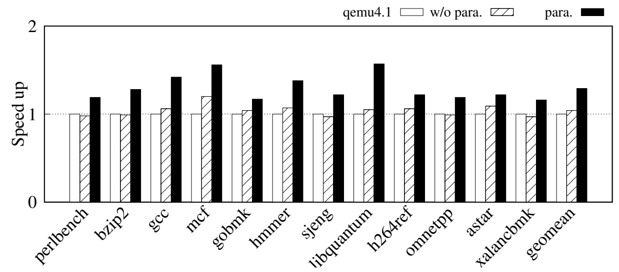

Fig. 11. Performance results with and without parameterization approach.

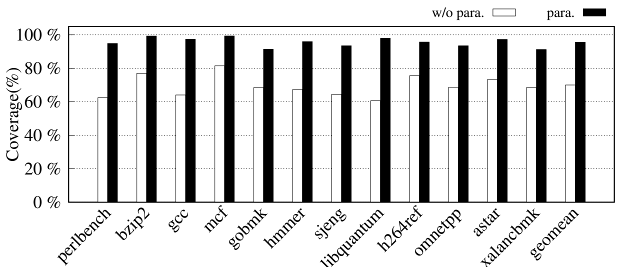

Fig. 12. The dynamic coverage *with* and *without* parameterization.

1) Performance Results: The performance (i.e., runtime) of each benchmark in SPEC CINT 2006 is normalized to its QEMU 4.1 performance. In Figure 11, *"w/o para."* means the original learning-based approach, and *"para."* means our parameterization approach. As the results show, our approach is quite effective in improving performance. It can achieve an average of 1.29X speedup compared to QEMU 4.1, and about 1.24X speedup on average compared to the state-of-the-art enhanced learning-based approach [16] without using parameterization.

To have a deeper understanding on the performance improvement, we collect dynamic coverage of the guest binaries using our parameterized approach. As the results in Figure 12 show, we can achieve 95.5% dynamic coverage on average using parameterization on learned translation rules, compared to only 69.7% without using it [16]. This shows that the translation rules have been greatly expanded to cover more guest binaries after parameterization is applied.

Since the rules are learned from the binary codes generated by an optimizing compiler, it is more likely for a learning-based DBT system to produce higher-quality host codes. Such high-quality host codes are automatically extended to the host codes generated by the parameterized rules. As more such rules are being used, better performance can be achieved. One metric to determine the quality of the translated code is the ratio of the translated host instructions to the guest instructions, i.e., the average number of host instructions needed to emulate one guest instruction after the translation. This metric is important because program execution time is directly proportionate to the number of instructions executed. As the results in Figure 13 show, our parameterization approach can generate higher-quality host codes compared to prior methods.

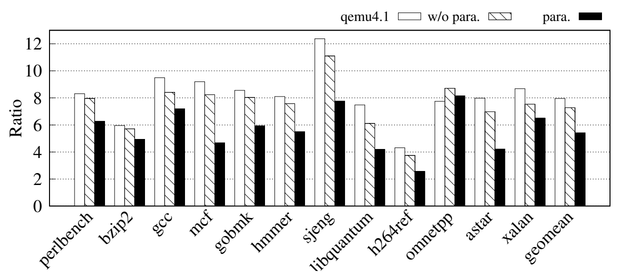

Fig. 13. The number of host instructions that one guest instruction is translated to.

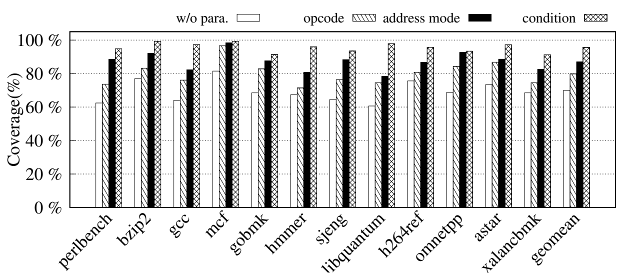

Fig. 14. The *dynamic coverage* improvement from the parameterization and the condition flags delegation.

On average, one guest instruction is translated into 5.66 host instructions by our approach, 7.51 host instructions using the existing learning-based approach, and 8.18 host instructions using QEMU 4.1.

To simplify the implementation, we leverage existing mechanisms in QEMU as much as possible. QEMU translated each guest instruction first into one or more pseudo-instructions, called TCG instructions. Each TCG instruction is then translated into one or more host instructions. Using a learning-based approach, one or several guest instructions are translated directly into host instructions according to the translation rules without going through TCG IR.

The translated host instructions are stored in a *"code cache"* for reuse. In this way, we need not translate guest instructions in loops or subroutines again each time they are emulated. In the code cache, the translated *host* code is organized according to the *basic blocks* in the *guest* binary. In addition, guest registers are emulated through an array in the host memory space. When a guest basic block is translated, some load (store) instructions are needed to retrieve (update) the guest register values from (to) their corresponding memory locations. Moreover, some additional instructions are needed to handle the control flow among different guest basic blocks, called *"stubs"* in QEMU. To give a more detailed account on those additional host instructions needed at the *basic block* level, we show some results in Table II.

As the data in Table II shows, using the parameterized translation rules (marked as *Rule translated*), 0.97 host instructions are needed to translate one guest instruction, as opposed to 3.49 host instructions in QEMU (marked as *QEMU translated*). The ratio of 0.97 reflects somewhat the fact that ARM uses RISC ISA and X86 uses more "powerful" CISC ISA, and the fact that QEMU goes through TCG IR before generating host binaries.

These instruction counts have not included those instructions needed to maintain a guest basic block in the code cache (i.e. stubs). When load/store instructions for maintaining guest register values are included (marked as *Data transfer* in the table), 2.02 such host instructions are needed on average for each guest instruction. In addition, when the stub code is included (marked as *Control code*), 2.68 such host instructions on average are needed per guest instruction.

In total, using the parameterized rules (marked as *Rule total*), we need 5.66 host instruction to translate one guest instruction, and QEMU (marked as *QEMU total*) needs 8.18 host instructions on average. Based on the above results, we can see our parameterization approach is more efficient than QEMU . Also, a lot of the overhead actually comes from the instructions needed to maintain basic blocks in code cache. They can be removed through several optimization techniques such as *chaining* guest basic blocks into code *traces*, and using global register allocation to eliminate load/store instructions needed before and after each basic block [7], [9], [23]. Those optimizations are beyond the scope of this paper.

2) Major Factors on Performance: As mentioned earlier, translation rules are parameterized along the two dimensions of opcode and addressing mode with the consideration of condition flags. Figure 14 shows the impact on performance and dynamic coverage from the opcode parameterization (marked as *"opcode"*), the addressing mode (marked as *addressing mode*), and the condition flags delegation (marked as *condition*). The baseline used for comparison is the enhanced learning-based DBT system [16].

As the results in Figure 14 show, *dynamic coverage* can be increased from 69.7% to 79.8% by the opcode parameterization, further increased to 87.0% from the addressing mode parameterization, and increased again to 95.5% from the condition flags delegation. Figure 15 shows the *performance improvement* from three factors. It is improved from 1.04X (the original learning-based approach) to 1.13X through the opcode parameterization, increased to 1.22X from the addressing mode parameterization, and further increased to 1.29X from the condition flags delegation compared to QEMU 4.1. From these results, we can see that the performance improvement and that of coverage are strongly correlated.

Furthermore, the extent to which the parameterization can impact on the performance and the dynamic coverage of each benchmark depends on the characteristics of the benchmark. For example, if more instruction types are used in an application, more performance improvement can be obtained from the opcode parameterization. Here, opcode parameterization contributes to 10.1% improvement on code coverage and 13% to performance improvement on average. But in *h264ref*, the opcode parameterization only achieves 5.1% and 2.8% improvement on coverage and performance, respectively, which is much less than the other programs. This is because

TABLE II
THE INSTRUCTION NUMBER OF THE HOST CODE TRANSLATED PER GUEST INSTRUCTION OF PARAMETERIZATION AND QEMU .

Benchmark	Rule translated	QEMU translated	Data transfer	Control code	Rule total	QEMU total
perlbench	1.15	3.19	2.07	3.06	6.28	8.31
bzip2	0.94	1.97	2.10	1.89	4.93	5.96
gcc	0.83	3.12	1.99	4.38	7.20	9.49
mcf	0.97	5.50	2.28	1.43	4.67	9.20
gobmk	1.01	3.64	1.95	2.97	5.93	8.56
hmmer	0.90	3.57	1.26	3.27	5.51	8.10
sjeng	1.38	5.98	2.87	3.52	7.77	12.37
libquantum	0.95	4.24	1.48	1.76	4.19	7.48
h264ref	0.74	2.48	1.02	0.80	2.56	4.31
omnetpp	0.79	0.39	3.09	4.27	8.15	7.75
astar	1.01	4.76	1.82	1.39	4.22	7.98
xalancbmk	0.85	3.02	2.27	3.39	6.51	8.68
Average	0.97	3.49	2.02	2.68	5.66	8.18

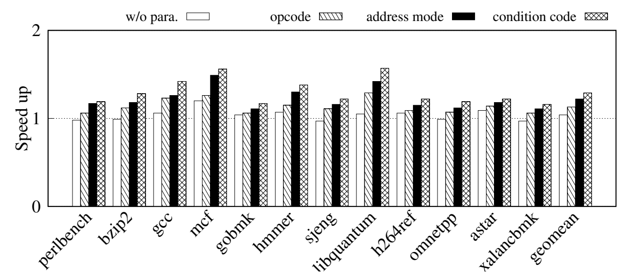

Fig. 15. The *performance* improvement from the parameterization and condition flags delegation.

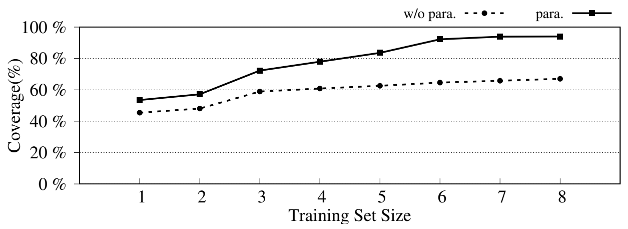

Fig. 16. The comparison of different training sets.

instruction types used in *h264ref* are much fewer than those in the other applications. More translation rules thus have less impact on its performance and coverage. As for condition flags, *libquantum* can improve both its coverage and performance significantly. This is because a loop dominated by an *eor* instruction cannot be translated due to different condition flags used. However, after parameterization and condition flags delegation, the rules can be applied.

Even though we can improve the dynamic coverage to more than 95%, there are still more than 4% of instructions that cannot be translated and need to be emulated. After a more in-depth analysis, we find that there are seven instructions: *push*, *pop*, *bl*, *b*, *mla*, *umla* and *clz* whose translation rules cannot be obtained through the learning process. The instruction *push*, *pop* and *bl* are function call/return-related instructions. The differences in the *application binary interface (ABI)*, in which ARM is register-based while X86 is stack-based, cause those instructions to be tied to a sequence of different instructions associated with ABI, e.g. different number of arguments. They thus cannot be formalized as well-defined translation rules. Similarly, the instruction *b* is always combined with some other instructions (e.g. *cmp* or *tst*) depending on the conditional statements. Hence, an individual *b* instruction cannot be learned through the proposed learning process. The instructions *mla*, *umla* and *clz* are special instructions in ARM, which have no corresponding instructions in X86.

Since there are only a very small number of such instructions, they can be added manually into the translation rules with very minimal engineering effort. Those individual instructions in the instruction sequences that are involved in ABIs can be translated individually using individual translation rules. In this way, 100% coverage can be achieved.

C. Analysis of Training Set

One benefit of the parameterization approach is that we can get more rules and achieve better performance results with a smaller training set. In this section, we evaluate how the training set affects the parameterization approach by shrinking the training set. We randomly select a certain number of benchmarks from SPEC CINT 2006 for training, and apply the harvested rules to the remaining ones. The reduced training sets go up from randomly selected 1 to 8 programs. For each training set, we collect the coverage results for both the parameterized and the non-parameterized approaches [16]. To minimize the variations due to randomness, the process is repeated 5 times (with different training sets of the same size), and the average results are used for comparison. As the results in Figure 16 show, the parameterization approach always gets a higher dynamic program coverage than the non-parameterized approach. With the sizes of the reduced training sets increasing, the dynamic program coverage also increases. It saturates at around 6 programs for both approaches. With a reduce training set of 8 randomly selected programs, the parameterized approach achieves about 95.5% dynamic coverage while the non-parameterized approach achieves only 69.7% dynamic coverage on average.

TABLE III
RULE NUMBER COMPARISON.

Approaches	**No. of Parameterized Rules**
Orig. learned rules	2724
Opcode para.	2401
Addressing mode para.	1805

Approaches	**No. Instantiated Rules**
Orig. learned rules	2724
Parameterization.	86423

D. Analysis of Parameterized Rules

The proposed parameterization approach has greatly increased the the number of applicable rules. Moreover, it avoids a large training set and training time. In this section, we analyze the rules obtained from the parameterization process. We first collect the data on the number of rules we can obtain from SPEC CINT 2006 after the parameterization. The number of parameterized rules should be smaller than the number of learned-rules because each parameterized rule represents several learned rules in the same subgroups (Section IV-B). As the data in Table. III show, there are 2724 learned rules from the existing learning-based method [16]. After the *opcode* parameterization, we have 2401 parameterized rules. After the *addressing mode* parameterization, the total number of parameterized rules becomes 1805.

Here, we only parameterize the learned rules with a *single* guest instruction. We did not parameterize the learned rules with instruction sequences, i.e. with more than one guest instruction in the sequence. Parameterizing instruction sequences will yield more rules, but will make the rule application slightly more complicated. We find that parameterizing learned rules with a single guest instruction is sufficient to increase the code coverage to more than 95% with minimal complexity in rule application. However, parameterizing guest instruction sequences will improve the performance further because they can produce more optimized host code sequences after translation.

Those parameterized rules will be instantiated and substituted with the corresponding opcode and addressing mode when they are applied during the translation and host code generation at runtime (Section IV-D). The total number of instantiated rules is determined by the number of opcodes and addressing modes that can be substituted in each parameterized rule and the total number of the parameterized rules. For the 1805 parameterized rules we obtain from SPEC CINT 2006, they can be instantiated to 86423 rules during the translation. Moreover, all of the integer instructions, except the 6 instructions mentioned earlier, are covered by those rules with a performance gain of 1.29X compared to QEMU 4.1.

VI. Related Work

There has been a lot of work done to improve the performance of the DBT systems [1]–[3], [9], [10], [19]–[21]. Those optimization techniques can mostly be used in conjunction with our parameterization approach to further improve DBT performance. In this section, we only discuss the work most related to the techniques that can improve the quality of the translated host binaries.

HQEMU [7] improves the quality of the translated binaries and their performance by translating the guest binary into the LLVM intermediate representation (IR). It then leverages the LLVM JIT compiler to generate higher quality binary code at runtime. It can improve DBT performance but can also incur significant runtime overhead if it is for short-running guest binaries. In addition, because of the lack of source-level information in the LLVM IR translated from the guest binary code, it is quite challenging to take full advantage of the LLVM JIT optimization. In contrast, the learning-based approach has already taken advantage of the source-level information by the optimizing compiler, which is done offline, to produce optimized host binary in the translation rules.

Another work in [26] leverages host hardware features by applying optimization to host binary code after the translation. Some recent work [6] optimizes dynamically-generated guest binary code. In [17], it uses the host hypervisor to support system functions that can only be emulated by software means. Some approaches [5], [8], [12] try to translate guest single-instruction-multiple-data (SIMD) instructions. Different from those approaches, most of which manually constructed translation rules, our approach uses a parameterized learning-based scheme to produce translation rules.

The learning-based approach was originally proposed in [16], [18]. Our parameterization approach significantly extends its capability and functionality to improve both code coverage and performance. It can generate translation rules that are difficult to learn even in a large training set, and can cut down the learning process substantially.

VII. Conclusion

In this paper, we present a parameterization approach for learning-based DBT system. It parameterizes the learned rules in two dimensions: opcode and addressing mode. Based on the experimental results from a prototype using SPEC CINT 2006 as the benchmarks, we find our approach can improve the dynamic coverage from 69.7% in the prior learning-based approach to 95.5%, and achieve a performance improvement of up to 1.49X with an average of 1.24X.

Acknowledgments

We are very grateful to our shepherd 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 Municipal Science and Technology Major Project (No. 2017SHZDZX01), Shanghai Technology Development and Entrepreneurship Platform for Neuromorphic and AI SoC, the Shanghai Research and Innovation Functional Program (No. 17DZ2260900), and a faculty startup funding of the University of Georgia.

References

[1] C.-J. Chang, J.-J. Wu, W.-C. Hsu, P. Liu, and P.-C. Yew, "Efficient memory virtualization for cross-isa system mode emulation," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '14. New York, NY, USA: ACM, 2014, pp. 117–128. [Online]. Available: http://doi.acm.org/10.1145/2576195.2576201

[2] E. G. Cota, P. Bonzini, A. Bennée, and L. P. Carloni, "Cross-isa machine emulation for multicores," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 210–220. [Online]. Available: http://dl.acm.org/citation.cfm?id=3049832.3049855

[3] A. D'Antras, C. Gorgovan, J. Garside, and M. Luján, "Low overhead dynamic binary translation on arm," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 333–346. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062371

[4] P. Feiner, A. D. Brown, and A. Goel, "Comprehensive Kernel Instrumentation via Dynamic Binary Translation," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150992

[5] S.-Y. Fu, D.-Y. Hong, Y.-P. Liu, J.-J. Wu, and W.-C. 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*, ser. LCTES 2017. New York, NY, USA: ACM, 2017, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/3078633.3081029

[6] B. Hawkins, B. Demsky, D. Bruening, and Q. Zhao, "Optimizing binary translation of dynamically generated code," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 68–78. [Online]. Available: http://dl.acm.org/citation.cfm?id=2738600.2738610

[7] D.-Y. Hong, C.-C. Hsu, P.-C. Yew, J.-J. Wu, W.-C. Hsu, P. Liu, C.-M. Wang, and Y.-C. Chung, "Hqemu: a multi-threaded and retargetable dynamic binary translator on multicores," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 2012, pp. 104–113.

[8] D.-Y. Hong, Y.-P. Liu, S.-Y. Fu, J.-J. Wu, and W.-C. Hsu, "Improving simd parallelism via dynamic binary translation," *ACM Trans. Embed. Comput. Syst.*, vol. 17, no. 3, pp. 61:1–61:27, Feb. 2018. [Online]. Available: http://doi.acm.org/10.1145/3173456

[9] C.-C. Hsu, P. Liu, J.-J. Wu, P.-C. Yew, D.-Y. Hong, W.-C. Hsu, and C.-M. 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*, ser. VEE '13. New York, NY, USA: ACM, 2013, pp. 23–32. [Online]. Available: http://doi.acm.org/10.1145/2451512.2451519

[10] N. Jia, C. Yang, J. Wang, D. Tong, and K. 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*, ser. VEE '13. New York, NY, USA: ACM, 2013, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2451512.2451516

[11] P. Kedia and S. Bansal, "Fast Dynamic Binary Translation for the Kernel," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 101–115. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522718

[12] J. Li, Q. Zhang, S. Xu, and B. Huang, "Optimizing dynamic binary translation for simd instructions," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 269–280. [Online]. Available: http://dx.doi.org/10.1109/CGO.2006.27

[13] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. 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*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 337–348. [Online]. Available: http://doi.acm.org/10.1145/2150976.2151012

[14] A. Mittal, D. Bansal, S. Bansal, and V. Sethi, "Efficient Virtualization on Embedded Power Architecture®Platforms," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 445–458. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451163

[15] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 475–486. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485963

[16] C. Song, W. Wang, P.-C. Yew, A. Zhai, and W. Zhang, "Unleashing the power of learning: An enhanced learning-based approach for dynamic binary translation," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 77–90. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/song

[17] T. Spink, H. Wagstaff, and B. Franke, "A retargetable system-level DBT hypervisor," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 505–520. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/spink

[18] W. Wang, S. McCamant, A. Zhai, and P.-C. 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*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 84–97. [Online]. Available: http://doi.acm.org/10.1145/3173162.3177160

[19] W. Wang, J. Wu, X. Gong, T. Li, and P.-C. Yew, "Improving dynamically-generated code performance on dynamic binary translators," in *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '18. New York, NY, USA: ACM, 2018, pp. 17–30. [Online]. Available: http://doi.acm.org/10.1145/3186411.3186413

[20] W. Wang, P.-C. Yew, A. Zhai, and S. McCamant, "A General Persistent Code Caching Framework for Dynamic Binary Translation (DBT)," in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '16. Berkeley, CA, USA: USENIX Association, 2016, pp. 591–603. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026959.3027013

[21] W. Wang, P.-C. Yew, A. Zhai, and S. McCamant, "Efficient and scalable cross-isa virtualization of hardware transactional memory," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 107120.

[22] W. Wang, P.-C. Yew, A. Zhai, S. McCamant, Y. Wu, and J. Bobba, "Enabling cross-isa offloading for cots binaries," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 319331. [Online]. Available: https://doi.org/10.1145/3081333.3081337

[23] Z. Wang, R. Liu, Y. Chen, X. Wu, H. Chen, W. Zhang, and B. Zang, "Coremu: A scalable and portable parallel full-system emulator," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 213–222. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941583

[24] Q. Yang, Z. Li, Y. Liu, H. Long, Y. Huang, J. He, T. Xu, and E. Zhai, "Mobile Gaming on Personal Computers with Direct Android Emulation," in *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '19. New York, NY, USA: ACM, 2019.

[25] 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*, vol. 28, no. 10, pp. 2939–2952, Oct 2017.

[26] X. Zhang, Q. Guo, Y. Chen, T. Chen, and W. 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*, ser. CGO '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 246–256. [Online]. Available: http://dl.acm.org/citation.cfm?id=2738600.2738631