A System-Level Dynamic Binary Translator using Automatically-Learned Translation Rules

Jinhu Jiang[†‡], Chaoyi Liang[†], Rongchao Dong[†], Zhaohui Yang[†],
Zhongjun Zhou[†], Wenwen Wang[#], Pen-Chung Yew[§], Weihua Zhang[†]
School of Computer Science, Fudan University, Shanghai, China[†]
Institute of Bigdata, Fudan University, Shanghai, China[‡]
School of Computing, University of Georgia, Athens, GA, USA[#]
Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA[§]
{jiangjinhu, 18302010035, 20212010113, 18302010057, 19212010010, zhangweihua}@fudan.edu.cn,
wenwen@cs.uga.edu, yew@umn.edu

Abstract—**System-level emulators have been used extensively for the design, debugging and evaluation of the system software. They work by providing a system-level virtual machine that can support a guest operating system (OS) running on a platform with the same or different native OS using the same or different instruction-set architecture. For such a system-level emulation, dynamic binary translation (DBT) is one of the core technologies. A recently proposed learning-based approach using automatically-learned translation rules has shown to improve DBT performance significantly with much higher quality translated code. However, it has only been used on *user-level* emulation, not *system-level* emulation.**

In applying this approach directly on QEMU for system-level emulation, we find it actually causes an unexpected performance degradation of 5% on average. By analyzing its main culprits in more detail, we find that the learning-based approach will by default use host registers to maintain the guest CPU states that include condition-code registers (or FLAG registers). In cases where QEMU needs to be involved (in which QEMU also needs to use the host registers), maintaining system states in the host registers for the guest, the host and QEMU *during* and *between* the context switches can cause undue overheads, if not handled carefully. Such cases include emulating system-level instructions, address translation and interrupts, which require the use of QEMU's helper functions. To achieve the intended performance improvement through better-quality code generated by the learning-based approach, we propose several optimization techniques that include reducing the overhead incurred in each context switch, the number of needed context switches, and better code scheduling to eliminate context switches. Our experimental results show that such optimizations can achieve an average of 1.36X speedup over QEMU 6.1 using SPEC CINT2006 and 1.15X on real-world applications in the system emulation mode.

I. Introduction

System-level emulators are an important tool for designing new system architectures, debugging binary codes and profiling application programs in a full system environment. A system-level emulator emulates the binary code of a guest operating system (OS) implemented on the guest instruction set architecture (ISA), and booted on a host with the same or a different ISA running the same or a different OS. In a system-level emulation, dynamic binary translation (DBT) is one of the core technologies. This technology has been used in many applications provided by VMware, Valgrind, QEMU [1], and Rosetta. In essence, a DBT system provides the capability of dynamically translating a guest binary code to run on a host with a different system environment at runtime.

Depending on usage scenarios, a DBT can emulate guest binaries at the *user* level or the *system* level. When at the *user* level, the DBT only translates the guest binaries and runs the translated binaries directly on the host OS without emulating detailed guest OS operations such as virtual address translation and system calls. While at the *system* level, the DBT needs to translate the entire guest execution environment that includes all guest OS operations. It thus provides a complete system-level emulation on top of the host environment.

A general dynamic binary translator, such as QEMU, provides a general framework that includes an intermediate representation (IR) as a common interface between the guest and the host binaries. Guest binaries are first translated to the QEMU IR, and then from the IR to host binaries in different ISAs, i.e., it is a ”many(ISAs)-to-many(ISAs)” binary translation. However, due to this two-step translation approach, each guest instruction will be translated into n IR instructions and each IR instruction to m host instructions, with a total of nxm host instructions. For example, [2] shows that a guest ARM instruction can be translated into 8.18 host x86 instructions on average. In addition, it requires significant engineering effort to manually create translation rules that translate each guest instruction to the IR, and then translate each IR instruction to the host binaries.

A learning-based DBT approach [2] [3] [4] was proposed recently to resolve those issues using automatically-learned translation rules for each pair of guest ISA and host ISA, i.e., an ”one(ISA)-to-one(ISA)” binary translation approach. These translation rules can be automatically learned from the optimized guest and host binary codes produced by compilers using the same source code. It is an ’one-step’ translation approach (i.e. without going through IR) based on high-quality translation rules with minimal engineering effort. However, this approach has only been applied to user-level emulation, and not yet to system-level emulation. Our recent experimental results show that, although the learning-based approach works well at the user level, it unexpectedly causes a 5% slowdown

979-8-3503-9509-9/24 © 2024 IEEE

after we apply it to a system-level emulation.

In this paper, we first analyze the new challenges in the system-level emulation. We find that, when it encounters system-level instructions as well as address translation and interrupts, which are common in a system mode emulation, the learning-based approach needs to switch to QEMU for various system support. Any DBT system, not just learning-based DBT systems, needs to maintain guest CPU states such as the content of general registers and condition codes/flags registers that are set implicitly. For example, QEMU uses a designated memory region to hold and maintain such guest CPU states. The learning-based approach, on the other hand, uses the *host registers* to hold and maintain guest CPU states by default. In cases where QEMU needs to be involved (in which QEMU also needs to use the host registers), maintaining system states in the host registers for the guest, the host and QEMU *during* and *between* the context switches can cause undue overheads. These overheads can offset the benefits derived from the learning-based approach if not coordinated and handled carefully.

To achieve a better performance, we propose several optimizations to reduce the overhead incurred *during* and *between* the context switches. One optimization is to delay the parsing of the guest CPU state. It can reduce the number of needed instructions to maintain the guest CPU state during each context switch. We also identify several common scenarios that can create rapid consecutive context switches and incur a substantial amount of overhead. They include (1) consecutive memory access instructions that require emulating the address translation in QEMU, (2) define-before-use translation blocks (TBs), and (3) consecutive context switching between the translated code and QEMU due to system calls, interrupts or instruction sequences not in the translation rules. We can consolidate and combine some of those episodes to reduce the number of context switches and their associated overheads. Further optimizations include doing a better code scheduling for guest instructions that define and use CPU states (e.g. condition codes/flags), which can reduce redundant instructions needed to maintain those CPU states.

To study the effectiveness and the performance improvement that the learning-based approach can achieve with those optimization techniques in a system-level DBT, we implemented a prototype on QEMU 6.1. We use SPEC CINT2006 to evaluate our design. Experimental results show that, compared to the QEMU running in a system mode, the learning-based approach without our proposed optimizations has an average of 5% slowdown. When all optimizations are applied, 48.83% of all operations that are related to maintaining guest CPU states can be eliminated. It can achieve a performance improvement of 1.36X over the baseline QEMU on average. In addition, we also use several real-world applications for evaluation. The results show that the learning-based approach achieves an average of 1.15X speedup.

In summary, this paper makes the following contributions:

- We apply the learning-based approach to system-level DBT, and propose several optimizations to reduce the overhead required to maintain CPU states *during* and *between* the context switches frequently encountered in the system mode.
- We implement the proposed learning-based system-level design and its optimizations in a prototype using QEMU.
- We conduct several experiments to evaluate the learning-based system-level emulator. The results show that we can achieve an average of 1.36X speedup over QEMU on SPEC CINT2006 and 1.15X on real-world applications with all of the proposed optimizations applied.

The rest of this paper is organized as follows. In Section II, we discuss the motivation and challenges in applying the learning-based approach to a system-level DBT. In Section III, we present the design of the learning-based system-level emulator and propose several optimizations to improve the coordination efficiency. In Section IV, we evaluate our implementation and show some experimental results. Section V includes some related work. Section VI concludes this paper.

II. Background & Motivation

In this section, we first give a brief introduction to the learning-based DBT approach, and then discuss the CPU state maintenance strategy in the learning-based DBT approach. Finally, we present the challenges of applying the learning-based approach to system-level DBT.

A. Learning-Based DBT Approach

There are three parts in the learning-based DBT approach: (1) learning of the translation rules, (2) parameterization of the learned rules, and (3) rules application [2] [3] [4].

In the learning phase, it uses an automated learning framework to generate high-quality translation rules. First, it compiles a source code into executable binaries of both the *guest* and the *host* architectures (e.g., ARM and x86) using popular compilers such as LLVM or GCC with the debug option turned on. Next, it uses the generated debugging information, such as the line numbers of the source code, to extract the semantically-equivalent code fragments in the two binaries, e.g., the binary sequences that correspond to the same source statement. These two code fragments form the basis of a translation rule since they are from the same source statement and are supposed to be semantically equivalent. It then uses a symbolic execution tool to perform a formal semantic-equivalence verification of the two code fragments. After the formal verification, the two code fragments (i.e., the guest and the host binary sequences) finally form a translation rule. This process can be iterated automatically using different training source codes to build a more comprehensive and complete translation rule set.

In the parameterization phase, the translation rules are parameterized to reduce the total number of rules in the rule set and achieve a higher coverage with a smaller training set. Instead of making each of these ALU-type instructions, such as `add`, `and`, and `or`, into a different translation rule, we can lump them together into *one* translation rule for all ALU-type instructions. In the rule-application phase during the binary translation, we first try to find a matched translation rule in the rule set. If we cannot find it, it will be switched to QEMU for emulation. And as mentioned earlier, this context switch will require the guest CPU states to be saved and restored. Such a learning-based approach can significantly improve DBT performance with translation rules learned from the native compilers. As its focus is on instruction-level translation, it is orthogonal to other optimizations such as memory-related optimizations [5], parallelism exploitation [6], and leveraging special host hardware features [7], [8].

B. CPU State Coordination

The guest CPU states contain all of the information needed to emulate guest binary codes. It includes the content of the *general-purpose registers*, status flags in the *condition code register* (CCR) and the *program counter* (PC). Some DBT systems, such as QEMU, maintain the guest CPU states in the *memory*. In QEMU, it maps the guest CPU states to a data structure in IR and maintains it in the memory. When executing the translated host binaries, it loads the guest CPU states from the memory. After the execution, it stores the latest guest CPU states back in the memory. Although this strategy is intuitive and easy to implement, it can generate a large number of memory operations and incur a significant overhead.

As the learning-based approach bypasses the IR and translates the guest binaries directly into the host binaries, it keeps the guest CPU states in the host CPU states as much as possible. This strategy can reduce the number of memory operations needed to maintain the CPU states. However, since the learning-based approach cannot achieve a 100% coverage, it will need to switch to QEMU to translate those instructions not covered in its rule set. The context switching between the translated binaries and QEMU requires additional overhead to maintain correct CPU states.

We use the example in Figure 1 to show how it works. Note that, like most DBT systems, QEMU maintains the translated binaries in a *code cache*. The unit of translated binaries organized in the code cache is a *basic block* of the *guest* binaries, marked as *TB1* and *TB2* in Figure 1. In this example, after executing the host binary in *TB1* we need to find and translate *TB2*, which will bring it back to QEMU.

At this point, the *guest* CPU state is being maintained in the host CPU registers as shown in the figure. After the host CPU is switched to QEMU, it will modify the host registers and corrupt the guest CPU state. Thus, at the end of *TB1*, we need to upload the guest CPU states to the memory locations where QEMU maintains the guest CPU states, as *Path 1* shows. After QEMU translates *TB2* and places the translated binary in the code cache, it needs to download the guest CPU states from the memory to the host registers where the translated binary in *TB2* maintains the guest CPU states as *Path 2* shows. We call the process of keeping the guest CPU states consistent during such a context switch *”CPU state coordination”*.

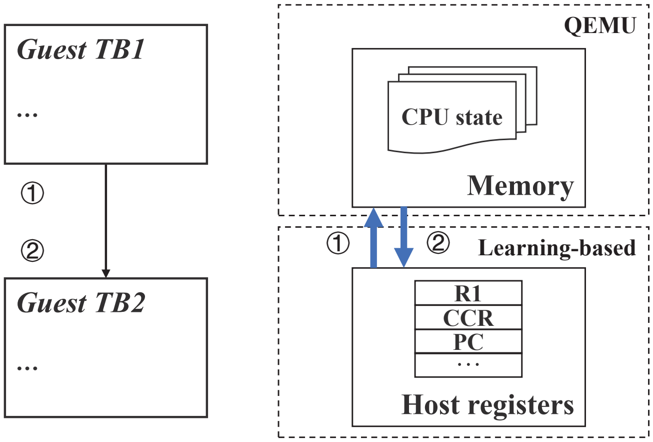


Fig. 1. CPU state maintenance and coordination.

C. Issues and Challenges

There are several challenges in CPU state coordination when we apply the learning-based approach directly in a system-level DBT. They can incur a large amount of overhead and lead to a 5% slowdown on QEMU as mentioned earlier.

System-level instructions. System-level instructions usually perform privileged operations in guest OS, which do not exist in user-level applications. As a result, the learning-based approach cannot automatically learn system-level instruction rules from user-level applications. It thus needs to use QEMU to translate the system-level instructions. QEMU uses a series of helper functions to emulate these instructions. For example, a privileged ARM instruction such as `vmsr`[1] is emulated by a QEMU helper function instead of translating into its corresponding host instruction sequence. When executing a helper function, it will context switch from the translated binaries in the code cache to QEMU. If the helper function needs to read or update guest CPU states, the guest CPU states can become inconsistent.

Figure 2 gives such an example. Assume the guest ISA is ARM. System-level instructions such as `vmsr` transfer the data between a VFP system register and an ARM register, in which VFP is a vector floating-point system register in an ARM processor. Assume that the instructions in the TB are all translated by the translation rules. When emulating the `vmsr` instruction, the learning-based approach invokes a QEMU-provided helper function. The helper function reads the ARM CPU state in the memory maintained by QEMU. However, this CPU state has expired because an earlier instruction ”`cmp al`” translated by rules will produce a new CPU state, and the learning-based approach maintains the latest guest CPU state in host registers. On the other side, the helper function for `vmsr` needs to maintain its own CPU state, which will overwrite the host registers and corrupt the guest CPU state stored there. The following instruction ”`add eq`” after the

[1] `vmsr` transfers the content of an ARM register to its VFP system register

helper function will have lost the guest CPU state produced by the ”`cmp al`” instruction.

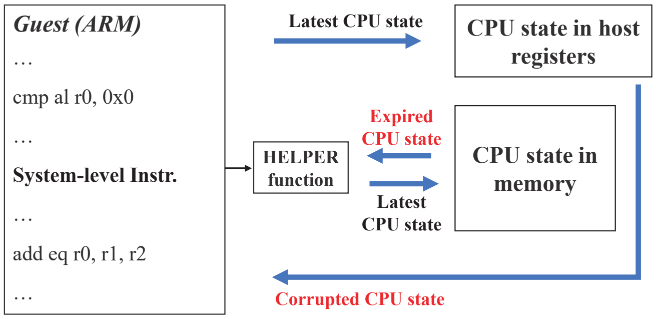


Fig. 2. An example of handling a system-level instruction.

Address translation. In a *user-level* DBT, it translates a memory address directly from its *guest virtual address* (GVA) to a *host virtual address* (HVA) by adding a fixed offset. However, in the *system-level* emulation, the DBT needs to emulate the guest *memory management unit* (MMU) for potential page faults. In this case, GVA is not necessarily mapped to a pre-determined HVA location. Therefore, it needs to be translated through an address translation process as shown in Figure 3.

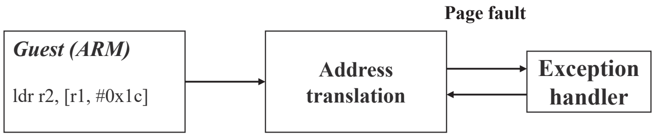


Fig. 3. Address translation.

The learning-based approach does not support the address translation process. When encountering a memory access instruction it will context switch to QEMU, and QEMU needs to use the host registers to carry out address translation thus corrupting the guest CPU states.

Interrupts. Figure 4 shows the interrupt handling mechanism in QEMU. Interrupts, such as I/O interrupts caused by the keyboard, are caught by the interrupt-check function (i.e., `check_interrupt()`) at the beginning of every TB as shown in the figure. The interrupt check function will invoke the corresponding interrupt handler to deal with the particular interrupt. QEMU has to translate and execute the interrupt handler provided by the *guest* OS. If system-level instructions are involved, a context switch from the translated code cache to QEMU is required.

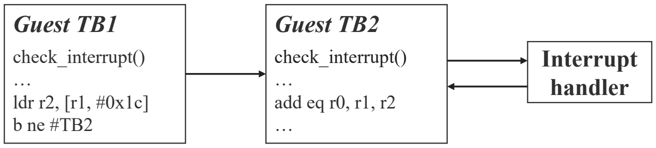


Fig. 4. System-level interrupt.

To estimate the overhead of such context switches, we collect at runtime the *dynamic numbers* of system-level instructions, memory access instructions, interrupt-check functions, and the total number of guest instructions executed in each application of SPEC CINT2006. We then calculate the occurrence frequency of each category per guest instruction, e.g., *# of system-level instructions / # of guest instructions*. The data are shown in Table I.

TABLE I
DISTRIBUTION OF THE THREE CATEGORIES THAT REQUIRE GUEST CPU STATE COORDINATION IN SPEC CINT2006.

Benchmark	System-level instr.	Memory instr.	Interrupt check
perlbench	0.28%	36.94%	19.64%
bzip2	0.28%	40.03%	14.24%
gcc	2.48%	29.90%	20.11%
mcf	0.45%	41.19%	20.53%
gobmk	0.25%	30.58%	17.53%
hmmer	0.09%	47.98%	5.18%
sjeng	0.17%	33.86%	17.84%
libquantum	0.09%	23.36%	9.19%
h264ref	0.13%	55.21%	9.15%
omnetpp	0.24%	22.54%	22.02%
astar	0.24%	31.42%	15.92%
xalancbmk	0.34%	23.81%	25.94%
GEOMEAN	0.25%	33.46%	15.12%

From the statistics, we can see that a context switch is required roughly every two guest instructions on average for QEMU at the system level. Most of the context switches are for memory access instructions (33.46%) and interrupt checks (15.12%). Only a small percentage is for system-level instructions (0.25%) in SPEC CINT2006.

III. Design & Optimizations

As mentioned earlier, the learning-based approach includes three phases: rule learning, parameterization, and rule application. CPU state coordination does not affect the rule learning and the parameterization phases. Thus, they will be the same as in the user-level DBT. However, in the rule-application phase, we propose an enhanced design with the optimizations mentioned earlier to reduce the overall context switch overheads.

A. Basic Guest CPU State Coordination

There are two types of guest CPU state coordination. One is the coordination needed when it switches from the code cache to QEMU shown as *Path 1* in Figure 1, we call it *sync-save* (as viewed from the perspective of the code cache). The other is to switch from QEMU back to the code cache shown as *Path 2* in Figure 1, called *sync-restore*. The needed operations are determined by the context that has the latest guest CPU state before it is switched to the other context.

To reduce overall overheads caused by context switching, a basic coordination scheme is shown in Figure 5. In the rule application phase, we first perform a scan on the guest TB to check each guest instruction. It marks the instructions in the TB, such as system-level instructions and `ld/st` instructions, that require guest CPU state coordination. We also identify what guest CPU states these instructions may read and/or write. Based on the information collected, the translation rule will insert codes to coordinate those CPU states when translating the corresponding instruction (see Figure 5).

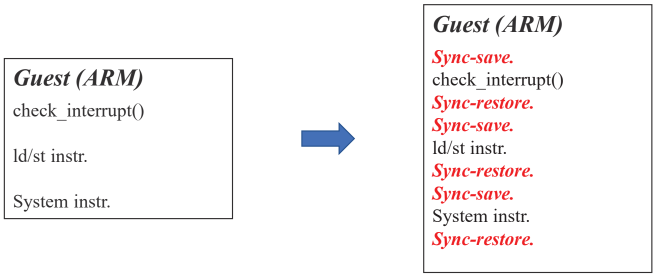


Fig. 5. A *basic* guest CPU state coordination.

We show the example of a system-level instruction such as a vmrs/vmsr in Figure 6. The translation rule will insert a *Sync-save* and a *Sync-restore* before and after the helper function used to emulate the guest system-level instruction. In the *Sync-save* before a vmrs instruction, it will upload the latest guest CPU state updated by the previous rule-translated *cmp al* instruction to QEMU, which allows the helper function to get the latest guest CPU state. Similarly, if the system-level instruction is a vmsr instruction, a *Sync-restore* will pass the latest guest CPU state from QEMU to the host registers after emulating this instruction. It allows the following instruction ”add eq” to have the latest guest CPU state in the host registers when we apply the learned translation rule on it.

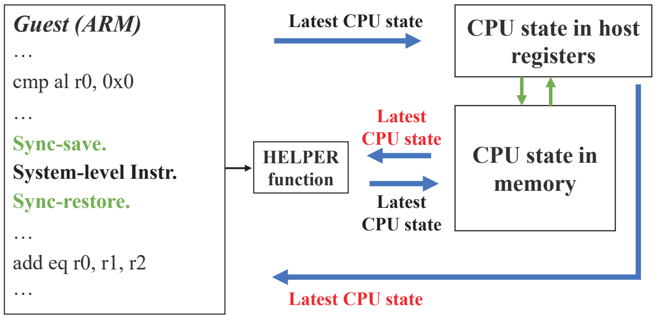


Fig. 6. An example of the guest CPU state coordination for a system-level instruction.

However, such a naive implementation will require very frequent guest CPU state coordination and incur a lot of runtime overhead. Thus, we propose three optimizations in the following subsections to reduce its overhead.

B. Coordination Overhead Reduction

In the three scenarios mentioned in Section II-C, only system-level instructions will update and modify guest CPU states. For address translation and interrupts, guest CPU state coordination is required only to prevent these states from being corrupted during a context switch. However, even if we only maintain those CPU states that will be modified in a context switch, our experimental results show that it still requires 14 instructions in each context switch, i.e., it only yields very modest overhead reduction.

In addition, in the rule-based binary translation[2], it may maintain several components of the CPU state in one register, but are maintained in separate memory locations by QEMU. A typical example is the bit-wise condition codes and flags, which are maintained in *one* condition-code register (CCR) in the rule-based translation since both x86 and ARM have such registers. But each condition code bit is kept in a different memory location in QEMU. We call this type of CPU state ”*one-to-many* CPU state”. In a Sync-save operation, it will need to parse the host CCR register and store each condition code in a different memory location using several store operations. However, if QEMU does not use them in its emulation, we can save the CCR register in one memory location using only one store operation and restore them afterward with only one load operation. In this way, we can reduce the total number of memory operations in Sync-save and Sync-restore operations.

Based on the above observation, depending on the types of the instructions collected in the lightweight parsing of a translation block, if a Sync-save operation involves an one-to-many state, we store the content of the host register that maintains the state to one memory location. In the case of the CCR register, unless QEMU needs to access any of the condition codes for emulation, we need not store them in separate memory locations designated by QEMU. As most of the Sync-save and Sync-restore operations are just to keep guest CPU state from being corrupted during context switches (see Table I), a significant number of memory operations can be eliminated this way.

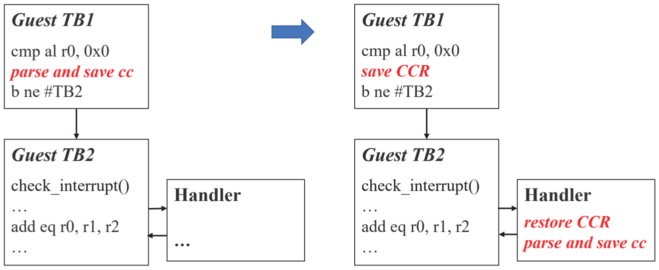


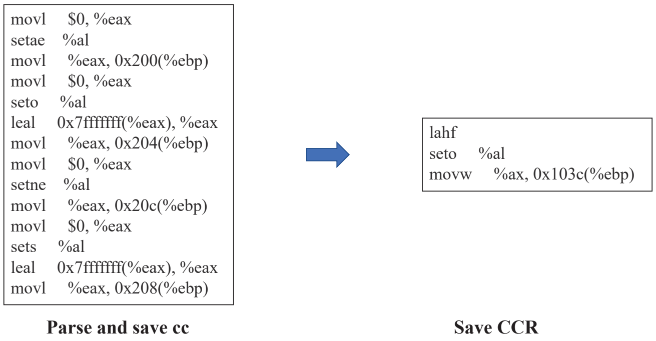
Fig. 7. Coordination overhead reduction.

Figure 7 shows an example that involves condition codes. It includes a check_interrupt() in TB2 that requires access to the condition codes. However, interrupts only occur very rarely. In this case, we save the content of CCR to a memory location at the end of TB1. If an interrupt has been triggered at the beginning of TB2, we restore the content of CCR, parse CCR and save the condition codes separately to their designated memory locations in QEMU. This is needed only when an interrupt occurs and the condition codes are actually needed. From our experimental results, we find

[2] Since the learning-based DBT uses translation rules for the binary translation, we also call it *rule-based* binary translation.

such interrupts occur very rarely in most applications. For example, it occurs only 0.0001% per guest instruction in SPEC CINT2006. A significant number of memory operations can thus be avoided.

Similar optimization can be applied to Sync-save and Sync-restore operations for system-level instructions and address translation as well. As the example shown in Figure 8, it needs about 14 instructions to parse the Eflags register (the CCR of x86) and save the condition codes to QEMU. After applying the optimization, only 3 instructions are actually needed, with a saving of (14-3)/14 = 78%.



```
movl    $0, %eax
setae   %al
movl    %eax, 0x200(%ebp)
movl    $0, %eax
seto    %al
leal    0x7fffffff(%eax), %eax
movl    %eax, 0x204(%ebp)
movl    $0, %eax
setne   %al
movl    %eax, 0x20c(%ebp)
movl    $0, %eax
sets    %al
leal    0x7fffffff(%eax), %eax
movl    %eax, 0x208(%ebp)
```

```
lahf
seto    %al
movw    %ax, 0x103c(%ebp)
```

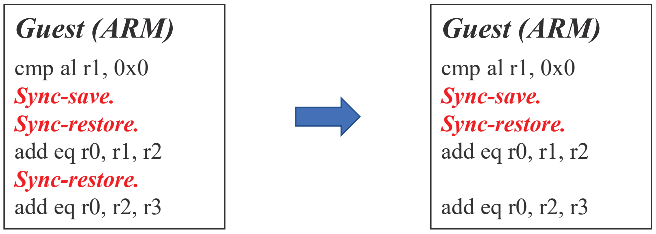
Fig. 8. Effect of coordination overhead reduction.

C. Coordination Elimination

Even after the overhead of each guest CPU state coordination is reduced by the optimization mentioned above, there are still cases where such coordination operations can be eliminated all together. We propose three such optimizations in this section.

1) Redundant Sync-restores Elimination: There are conditionally-executed instructions in some ISAs, which are executed based on whether the condition specified by the instruction is satisfied or not. For example, `add eq` is a conditional instruction in ARM-v7 that depends on the condition code `Z`. Only when the condition code `Z` is set, this instruction will be executed. In a DBT system, when a *guest* conditional instruction is translated, a *host* comparison instruction (e.g., `cmpl` in x86) is used to determine whether the current condition is satisfied or not. This will cause the guest CPU state maintained in the host registers to be corrupted. In the rule-based approach, as a host comparison instruction will change its CPU state that may be inconsistent with the semantics of its corresponding guest instruction, it is allowed only in a *constrained rule* [3]. The basic rule-based design will insert a Sync-restore after the comparison instruction to maintain the correct guest CPU state. However, if we encounter consecutive conditional instructions that depend on the same condition, we only need to restore the guest CPU state once at the first conditional instruction, and the remaining instructions can be translated normally without the additional comparison instructions and Sync-restore operations. To do this, in each TB, it first checks if there are such instructions that update the CPU state and the conditional instructions that depend on the updated CPU state in the TB. Next, it keeps the first Sync-restore before the first instruction in the TB that uses the CPU state, and eliminates the other Sync-restores until it reaches an instruction that requires a Sync-save operation or reaches the end of the TB.

As the example shown in Figure 9, we assume that after the instruction ”`cmp al`”, some situation (e.g., a system-level instruction) requires the Sync-save operation to save the CPU state to QEMU. At the same time, the next few instructions ”`add eq`” need to use the CPU state. The rule-based translation will have a Sync-restore for each of such instructions. But in this case, only the first Sync-restore is needed, and the rest of the Sync-restores and the translated comparison instructions can be eliminated. Through this optimization, a significant amount of coordination overhead due to the rule-based translation can be eliminated.



Guest (ARM)
cmp al r1, 0x0
Sync-save.
Sync-restore.
add eq r0, r1, r2
Sync-restore.
add eq r0, r2, r3

Guest (ARM)
cmp al r1, 0x0
Sync-save.
Sync-restore.
add eq r0, r1, r2

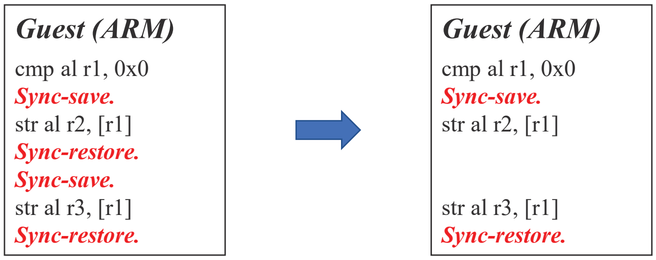
add eq r0, r2, r3

Fig. 9. Coordination-restore optimization.

2) Optimization for Consecutive Memory Operations: In the case of consecutive memory-access instructions, we can use a similar coordination elimination scheme to reduce redundant coordination. Because of the need to emulate address translation in QEMU at the system level, a Sync-save operation is inserted before and a Sync-restore after each memory access instruction during the rule-based translation. Apparently, if there are consecutive memory-access instructions in a TB, the intermediate coordination among those instructions can be removed. In this case, it first checks if it has a sequence of consecutive memory-access instructions in the TB. If it has, it keeps the Sync-save operation before the first memory-access instruction and the Sync-restore operation after the last memory access instruction in the sequence. The rest of the intermediate Sync-save and Sync-restore operations in the sequence can be removed.

As the example shown in Figure 10, there are two consecutive memory-access instructions `str` after the instruction ”`cmp al`”. The rule-based translation will insert two pairs of coordination in this situation. After applying the redundant coordination elimination, only a pair of Sync-save and Sync-restore are needed.

3) Inter-TB Optimization: As the unit of translation in a DBT is a guest TB, it needs to save the CPU state at the end of each TB and switch back to QEMU because the following TB may yet to be translated, or it may need to do a `check_interrupt` before the next TB. However, *block chaining* is a common optimization [9] [10] that chains



```
cmp al r1, 0x0
Sync-save.
str al r2, [r1]
Sync-restore.
Sync-save.
str al r3, [r1]
Sync-restore.
```

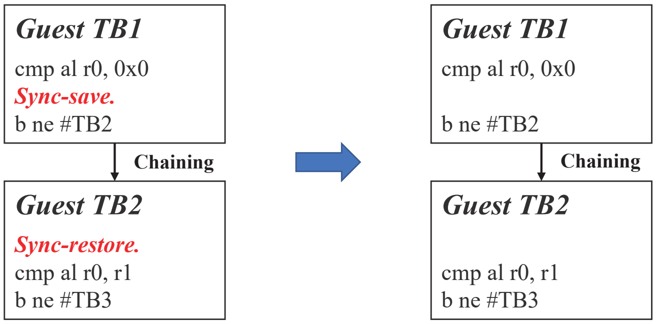
```
cmp al r1, 0x0
Sync-save.
str al r2, [r1]


str al r3, [r1]
Sync-restore.
```

Fig. 10. Optimization on consecutive ld/st instructions.

multiple TBs in the code cache together without context switching back to QEMU after each TB.

In this case, if the *first* guest instruction of the next TB will update a CPU state without using the CPU state defined in the previous TB, we will need neither a Sync-save operation at the end of the previous TB nor a Sync-restore at the beginning of the current TB. After an in-depth analysis of the execution flow in application codes, we find that if a series of TBs are chained together such an execution flow can be analyzed. In other words, in a code cache that uses block chaining similar to one used in QEMU, there is an opportunity to eliminate the Sync-save at the end of the current TB and the Sync-restore operation at the beginning of the next TB if the next TB will update its CPU state before using it.

Based on the above observation, we propose an inter-block optimization. It first checks if the current TB will jump to a TB in the code cache via block-chaining. Next, for each CPU state that needs to be coordinated, check the next TB to see if there is an instruction using the CPU state before it is updated by an earlier instruction in the TB. If not, we can omit the Sync-save operation at the end of the current TB and the Sync-restore operation at the beginning of the next TB.



```
cmp al r0, 0x0
Sync-save.
b ne #TB2

Sync-restore.
cmp al r0, r1
b ne #TB3
```

```
cmp al r0, 0x0

b ne #TB2


cmp al r0, r1
b ne #TB3
```

Fig. 11. Inter-TB optimization.

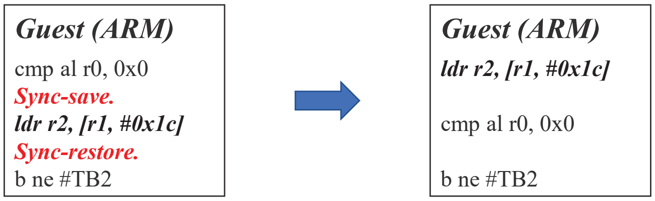
Figure 11 shows such an example. After the rule-based translation, the instruction ”`cmp al`” in TB1 will update the condition codes. It needs to do a Sync-save operation for the condition codes to be used in the following TB. In addition, the following TB2 needs to do a Sync-restore operation to obtain the correct condition codes. After traversing the block chain, it is found that TB1 jumps to TB2 in the chain. Furthermore, the instruction ”`cmp al`” in TB2 updates the condition codes before the instruction ”`b ne`” uses the condition codes. We can thus eliminate the Sync-save operation in TB1 and the Sync-restore operation in TB2.

D. *Instruction Scheduling*

We can also use instruction scheduling to reduce more redundant coordination. In this section, we present two scenarios, called *define-before-use scheduling* and *interrupt scheduling*, that can further reduce such coordination.

1) Define-Before-Use Scheduling: Within a TB, the instruction that uses the CPU state such as condition codes may be several instructions behind the instruction that updates such CPU state. There may be other instructions in between also. Some of them may even be system-level instructions or ld/st instructions that require QEMU's intervention. In the rule-based translation, a Sync-save operation will be inserted after the ”update/define” instruction and a Sync-restore will be inserted before the ”use” instruction. But, if there is no instruction in between that is data-dependent on either instruction, we can schedule the two instructions next to each other to avoid a CPU state coordination. We call such an instruction scheduling scheme the *define-before-use scheduling* scheme.

Figure 12 shows such an example. In this example, the instruction ”`cmp al`” updates the condition codes, and the instruction ”`b ne`” uses them. During the rule-based translation, it will insert a Sync-save operation and a Sync-restore operation before and after the memory-access instruction `ldr`. However, the `ldr` instruction is not dependent on the ”`cmp al`” instruction nor the ”`b ne`” instruction. By scheduling both instructions together after the `ldr` instruction, there is no longer a need to coordinate these condition codes before and after the `ldr` instruction.



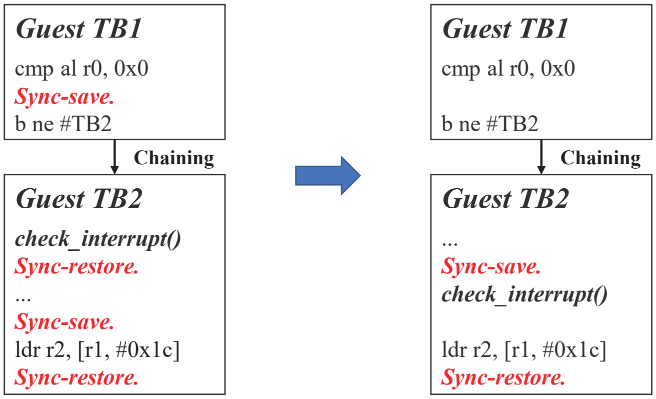
```
cmp al r0, 0x0
Sync-save.
ldr r2, [r1, #0x1c]
Sync-restore.
b ne #TB2
```

```
ldr r2, [r1, #0x1c]

cmp al r0, 0x0

b ne #TB2
```

Fig. 12. Define-before-use scheduling.

2) Interrupt-driven Scheduling: QEMU handles interrupts at the system level by inserting an interrupt-check function at the beginning of each TB. In theory, we can place the interrupt-check function in any other location in the TB. In the rule-based translation scheme, we insert CPU state coordination before and after each interrupt-check function. If the TB has memory-access instructions, we also need such coordination for each such instruction to prevent possible inconsistencies. Similar to the instruction scheduling scheme described earlier, if we can schedule the interrupt-check function close to the memory-access instructions, we can eliminate those redundant coordination. As the memory-access instructions appear quite frequently while interrupts rarely occur, there are ample opportunities to cut down such redundant coordination. We call such an approach *interrupt-driven scheduling*. It is particularly effective if block chaining is applied to TBs in the code cache.

Figure 13 shows such an example. Initially, CPU state coordination operations will be inserted both at the interrupt-check function and the memory-access instruction `ldr`. In fact, we can move the interrupt-check function to the front of the memory-access instruction `ldr`. This scheduling will not affect the interrupt handling and can reduce the coordination from two pairs to one.



```
Guest TB1
cmp al r0, 0x0
Sync-save.
b ne #TB2

Guest TB2
check_interrupt()
Sync-restore.
...
Sync-save.
ldr r2, [r1, #0x1c]
Sync-restore.
```

```
Guest TB1
cmp al r0, 0x0

b ne #TB2

Guest TB2
...
Sync-save.
check_interrupt()

ldr r2, [r1, #0x1c]
Sync-restore.
```

Fig. 13. Interrupt scheduling.

E. Optimization Interaction

The optimizations described above may change the corresponding host basic blocks in different ways. Therefore, we set different priorities among these optimizations and coordinate them for the best results. Redundant sync-restores elimination and optimization for consecutive memory operations mentioned in III-C, whose trigger conditions are orthogonal, simply reduce intra-block coordination and are applied first. Then, if the conditions of inter-TB optimization mentioned in III-C are met, we apply it to remove the coordination at the end of the block and the beginning of the next block. Otherwise, we use the coordination overhead reduction mentioned in III-B at the end of the block. Finally, we apply the instruction scheduling, changing the instruction order and further eliminating coordination. Note that this optimization order will not activate any previous optimizations.

IV. Evaluation

In this section, we evaluate our design and try to answer the following questions: (1) Using the rule-based translation approach at the system level with the proposed optimization to reduce redundant guest CPU state coordination, how much performance improvement can we achieve compared to state-of-the-art systems like QEMU 6.1? (2) How do various optimizations proposed in the paper affect the overall performance? (3) How does the performance of our approach compare to that of the native execution? (4) What about its performance improvement for real applications?

A. Experimental Setup

We have implemented a system-level rule-based DBT prototype based on QEMU 6.1. We take ARM-v7 as the guest ISA and Intel x86 as the host ISA. The translation rules used are the same parameterized translation rules used in [2]. The prototype is run on an Intel Xeon E5-2680 v4 machine with 2 cores, 56 threads and 126GB DRAM. The host OS is a 32-bit Ubuntu 14.04 with Linux 3.13. The guest OS is a 32-bit unmodified Linux system with kernel 4.4.0. We compile the SPEC CINT2006 using GCC-4.8 with -O2 optimization level and statically linking, and run the *ref* input of SPEC CINT2006 [3] on the guest OS. To better understand the performance of the rule-based approach, we also evaluate several real-world applications that include *memcached, sqlite, fileIO, untar* and *cpu-prime*, which are widely used in other system research [11], [12]. We run each benchmark ten times and take the average to reduce the effect of fluctuation. To measure the performance speedup, we use the execution time on *unmodified* QEMU 6.1 as the baseline.

B. Overall Performance

To study the effectiveness of our approach, we collect the performance data on QEMU 6.1 that includes the unmodified QEMU 6.1, the rule-based implementation of the system level on QEMU 6.1, and its optimized version with the three optimizations described in Section III (marked as "Full Opt." in the following figures).

As the results in Figure 14 show, the un-optimized rule-based implementation of QEMU 6.1 has a 5% slowdown compared to QEMU 6.1, i.e., it is actually slower than QEMU 6.1 running in system mode. The main reason is the CPU state coordination overhead explained in Section III. However, after the three optimizations are applied to reduce coordination overheads, the optimized rule-based QEMU 6.1 running at the system level can achieve a 1.36X speedup.

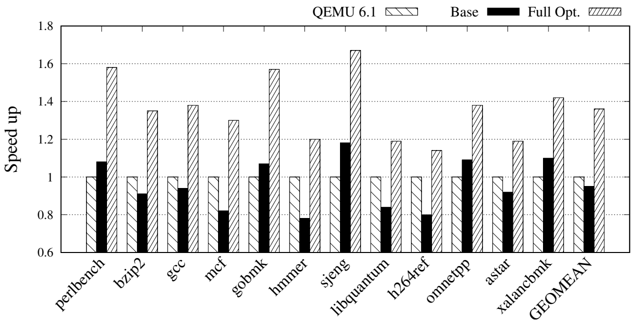


Fig. 14. Performance of SPEC CINT2006 running in system mode on unmodified QEMU 6.1, the un-optimized rule-based implementation of QEMU 6.1, and the optimized version of the rule-based implementation.

[3] The rule-based approach also supports the translation of floating-point instructions. Due to the space constraint, the floating-point applications in SPEC 2006 are not listed here. When these applications are included, our approach can achieve an average of 1.92X speedup over QEMU 6.1, instead of 1.36X speedup for only SPEC CINT2006.

Based on the data in Table I, an average of 48.83% guest instructions (which include system-level instructions, memory-access instructions and interrupt checks) will require CPU-state coordination operations. Moreover, each coordination operation will introduce around 14 host instructions. From Table I, for some benchmarks such as *mcf* and *h264ref*, the percentages of instructions that require such coordinations run as high as 62.17% and 64.49%, respectively. Their performance also suffers the most compared to other benchmarks as shown in Figure 14.

After the three optimizations are applied to remove redundant coordination operations, the percentage of instructions that require coordination is reduced to 24.61%, and the number of host instructions required in each coordination operation also goes down to only around 3 host instructions (as shown in Figure 8). To further understand the effect of the rule-based approach, we also collect the average number of host instructions needed to translate a guest instruction. The data are shown in Figure 15. As the data show, QEMU 6.1 in system mode requires an average of around 17.39 host instructions for each guest instruction, while the optimized rule-based implementation requires an average of 15.40 host instructions - a reduction of around 11.44%.

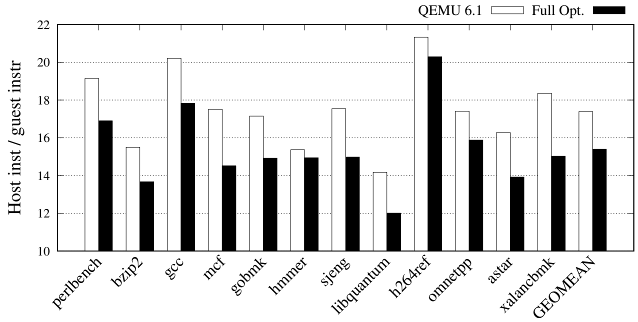


Fig. 15. Average number of host instructions needed to translate a guest instruction in un-modified QEMU 6.1, and in the optimized rule-based implementation.

To identify the performance bottleneck, we count the number of instructions in the host basic blocks and group the instructions by their functionality. Based on our analysis, one of the major bottlenecks is in the address translation. Since QEMU needs to emulate MMU behavior for each memory access in system mode, it involves about 20 host instructions for each translated memory instruction on average. This shows that the address translation incurs very high overheads, and it will be the focus for further optimization in our future work.

C. Impact of Coordination Optimizations

To understand the performance impact of each coordination optimization, we evaluate the *cumulative* performance improvement after adding each optimization. The results are shown in Figure 16.

In the figure, '*Base*' marks the performance of the un-optimized version as described in Section III-A. '+ *Reduction*' marks the performance after adding the optimization to reduce the number of host instructions in a coordination operation as described in Section III-B. '+ *Elimination*' marks the performance after further adding the optimization that eliminates redundant coordinations as described in Section III-C. '+ *Scheduling*' marks the performance after further applying instruction scheduling as described in Section III-D. The baseline is the performance of the unmodified QEMU 6.1.

As the data show, it achieves 1.22X speedup after the reduction optimization is applied. After adding the optimization that eliminates redundant coordination operations, the cumulative performance improvement is 1.30X. When all optimizations are applied, we achieve 1.36X overall speedup.

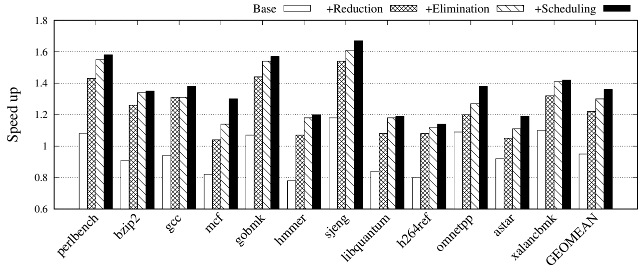


Fig. 16. Cumulative performance improvement after adding each proposed optimization.

To further understand the performance impact of these three optimizations, we also calculate the average number of host instructions needed for coordination per guest instruction. It is calculated by the following formula:

$$sync\_instr\ per\ guest\_ins = \frac{sync\_num * sync\_overhead}{guest\_num}$$

In the formula, *sync_num* is the total number of coordination operations, *sync_overhead* is the average number of host instructions used in a coordination operation, and *guest_num* is the total number of translated guest instructions. The results are shown in Figure 17.

Because we can reduce the number of host instructions in a coordination operation from 14 instructions to 3 instructions as shown in Figure 8, the average number of host instructions for coordination per guest instruction is reduced from 8.36 to 1.79. After the elimination of redundant coordination operations, the number of host instructions for coordination per guest instruction is further reduced to 1.33. When the instruction scheduling is finally applied, that number is eventually dropped to 0.89.

D. Comparison to Native Execution

The slowdown factor of a system-level emulation compared to the native execution of a program is an important factor in designing an emulator. We collect such performance data for both un-modified QEMU 6.1 and our rule-based optimized version of QEMU 6.1. The data are shown in Figure 18. Compared to QEMU 6.1, our rule-based optimized version of QEMU 6.1 achieves an average slowdown of 13.83X while QEMU 6.1 has an average of 18.73X slowdown.

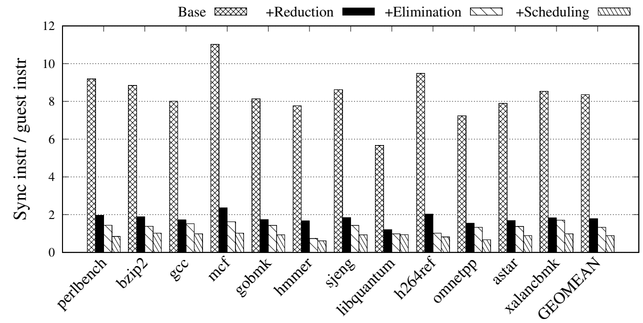


Fig. 17. Average number of host instructions per guest instruction for coordination after three optimizations are applied.

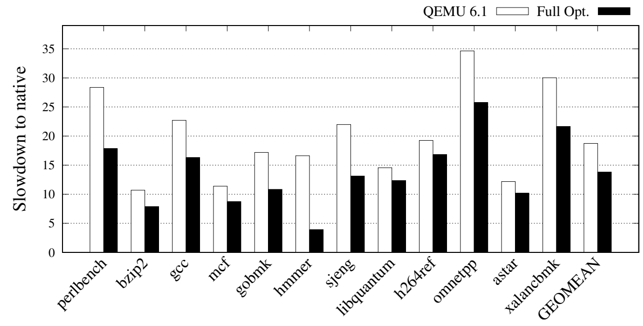


Fig. 18. Slowdown factors of the system-level emulation on un-modified QEMU 6.1 and the fully-optimized rule-based QEMU 6.1 compared to the native execution using SPEC CINT2006 (lower is better).

E. Performance on Real-World Applications

To better understand the performance of optimized rule-based approach at the system level, we evaluate the performance with several real-world applications. The real-world applications consists of *Memcached, Sqlite, FileIO, Untar* and *CPU-prime*. As the results in Figure 19 show, our design can achieve an average of 1.15X speedup over QEMU 6.1. In these applications, *FileIO* and *Untar* are IO-bound applications and *Memcached* is a network application. Since a lot of execution time is spent on IO or network, we can only achieve a speedup of 1.08X, 1.09X and 1.13X, respectively.

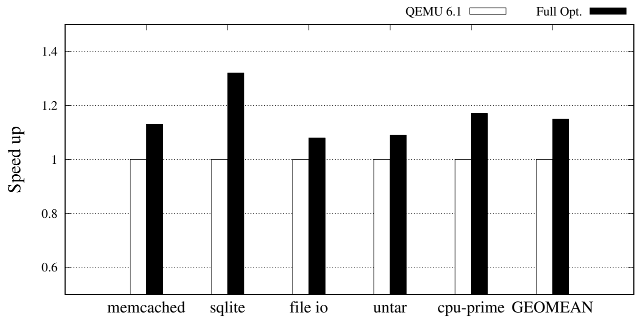


Fig. 19. Speedup of real-world applications on optimized rule-based QEMU 6.1 compared to un-modified QEMU 6.1.

V. Related Work

DBT systems have attracted extensive research. Prior work includes instruction translation and optimization [2]–[4], [13]–[16], system-level translation [1], [6], [7], [17], memory-access instruction optimization [5], [18], [19], indirect-branch optimizations [20]–[23], translation of architecture-specific instructions such as SIMD instructions [24]–[27], translation of atomic instructions [28], [29], and more. In this paper, we mainly focus on system-level DBT systems.

For system-level optimizations, [5] speeds up the memory address translation using embedded shadow page tables to do a direct mapping between a guest virtual address to its host physical address. [30] proposes a parallel system-level DBT emulator using a separate thread to optimize the translated code. Qlet [31] is a cross-ISA system-level instrumentation tool and several techniques is used to improve its performance. COREMU [6] uses one QEMU instance to emulate a multi-core system with a lightweight library for communication. By leveraging multi-core platforms and the optimizations in LLVM, HQEMU [10] proposed a parallel DBT system. The kernel-level binary translation mechanism in [32] achieves a near-native performance. [7], [8] translated binaries between ARM and x86 by utilizing host hardware features. The work in [17], [33], [34] takes advantage of hardware features to support system-level binary translation. Captive [17] is a retargetable system-level DBT hypervisor. It combines both offline and online optimizations running in a virtual bare-metal environment to deliver performance improvement.

Our work extends the rule-based DBT and applies it to the system-level DBT. It focuses on improving the quality of instruction translation, and can be combined with other optimizations applied to system-level DBTs to further improve performance, such as memory optimizations [5], parallelism [6] and hardware feature-based optimizations [7], [8].

VI. Conclusion

The rule-based approach using an automatic learning process to learn translation rules has shown to be effective in a DBT such as QEMU at the *user level*. To apply this approach to the *system level*, this paper presents a basic design to coordinate CPU states embedded in the guest and the host instructions when switching between the execution from the code cache and the emulation in QEMU. We address the issues critical to such a design and propose several optimization strategies to reduce such coordination overhead. We also implement a prototype based on QEMU 6.1 to demonstrate the feasibility of such an approach. The experimental results show that our design is quite efficient. Compared to QEMU 6.1, our fully optimized system can achieve an average of 1.36X speedup on SPEC CINT2006 and an average of 1.15X on real-world applications.

VII. Acknowledgements

We appreciate the anonymous reviewers for their valuable feedback and comments. This work is supported by the National Natural Science Foundation of China (No. 62141211).

References

[1] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. USENIX, 2005, pp. 41–46. [Online]. Available: http://www.usenix.org/events/usenix05/tech/freenix/bellard.html

[2] J. Jiang, R. Dong, Z. Zhou, C. Song, W. Wang, P. Yew, and W. Zhang, "More with less - deriving more translation rules with less training data for dbts using parameterization," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*. IEEE, 2020, pp. 415–426. [Online]. Available: https://doi.org/10.1109/MICRO50266.2020.00043

[3] C. Song, W. Wang, P. 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 2019, Renton, WA, USA, July 10-12, 2019*, D. Malkhi and D. Tsafrir, Eds. USENIX Association, 2019, pp. 77–90. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/song

[4] W. Wang, S. McCamant, A. Zhai, and P. 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 2018, Williamsburg, VA, USA, March 24-28, 2018*, X. Shen, J. Tuck, R. Bianchini, and V. Sarkar, Eds. ACM, 2018, pp. 84–97. [Online]. Available: https://doi.org/10.1145/3173162.3177160

[5] C. Chang, J. Wu, W. Hsu, P. Liu, and P. Yew, "Efficient memory virtualization for cross-isa system mode emulation," in *10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '14, Salt Lake City, UT, USA, March 01 - 02, 2014*, M. Hirzel, E. Petrank, and D. Tsafrir, Eds. ACM, 2014, pp. 117–128. [Online]. Available: https://doi.org/10.1145/2576195.2576201

[6] 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 SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, C. Cascaval and P. Yew, Eds. ACM, 2011, pp. 213–222. [Online]. Available: https://doi.org/10.1145/1941553.1941583

[7] A. D'Antras, C. Gorgovan, J. D. Garside, J. Goodacre, and M. Luján, "Hypermambo-x64: Using virtualization to support high-performance transparent binary translation," in *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017*. ACM, 2017, pp. 228–241. [Online]. Available: https://doi.org/10.1145/3050748.3050756

[8] A. D'Antras, C. Gorgovan, J. D. 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, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, A. Cohen and M. T. Vechev, Eds. ACM, 2017, pp. 333–346. [Online]. Available: https://doi.org/10.1145/3062341.3062371

[9] C. Wang, S. Hu, H. Kim, S. R. Nair, M. B. Jr., Z. Ying, and Y. Wu, "Stardbt: An efficient multi-platform dynamic binary translation system," in *Advances in Computer Systems Architecture, 12th Asia-Pacific Conference, ACSAC 2007, Seoul, Korea, August 23-25, 2007, Proceedings*, ser. Lecture Notes in Computer Science, L. Choi, Y. Paek, and S. Cho, Eds., vol. 4697. Springer, 2007, pp. 4–15. [Online]. Available: https://doi.org/10.1007/978-3-540-74309-5_3

[10] D. Hong, C. Hsu, P. Yew, J. Wu, W. Hsu, P. Liu, C. Wang, and Y. Chung, "HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores," in *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012, San Jose, CA, USA, March 31 - April 04, 2012*, C. Eidt, A. M. Holler, U. Srinivasan, and S. P. Amarasinghe, Eds. ACM, 2012, pp. 104–113. [Online]. Available: https://doi.org/10.1145/2259016.2259030

[11] K. Huang, F. Zhang, C. Li, G. Niu, J. Wu, and T. Liu, "BTMMU: an efficient and versatile cross-isa memory virtualization," in *VEE '21: 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Virtual USA, April 16, 2021*, B. L. Titzer, H. Xu, and I. Zhang, Eds. ACM, 2021, pp. 71–83. [Online]. Available: https://doi.org/10.1145/3453933.3454015

[12] J. Chen, D. Li, Z. Mi, Y. Liu, B. Zang, H. Guan, and H. Chen, "Duvisor: a user-level hypervisor through delegated virtualization," *CoRR*, vol. abs/2201.09652, 2022. [Online]. Available: https://arxiv.org/abs/2201.09652

[13] F. Salgado, T. Gomes, S. Pinto, J. Cabral, and A. Tavares, "Condition codes evaluation on dynamic binary translation for embedded platforms," *IEEE Embed. Syst. Lett.*, vol. 9, no. 3, pp. 89–92, 2017. [Online]. Available: https://doi.org/10.1109/LES.2017.2718531

[14] X. Liu, R. Zhao, J. Pang, M. Yin, L. Bai, and W. Chen, "A flag simulation strategy based on fusion of semantic trees in binary translation," in *10th International Conference on Fuzzy Systems and Knowledge Discovery, FSKD 2013, Shenyang, China, July 23-25, 2013*, J. Chen, X. Wang, L. Wang, J. Sun, and X. Meng, Eds. IEEE, 2013, pp. 1070–1074. [Online]. Available: https://doi.org/10.1109/FSKD.2013.6816355

[15] C. Chu, Y. Zheng, H. Guan, and A. Liang, "A two-phase optimization approach for condition codes in a machine adaptable dynamic binary translator," in *CSIE 2009, 2009 WRI World Congress on Computer Science and Information Engineering, March 31 - April 2, 2009, Los Angeles, California, USA, 7 Volumes*, M. Burgin, M. H. Chowdhury, C. H. Ham, S. A. Ludwig, W. Su, and S. Yenduri, Eds. IEEE Computer Society, 2009, pp. 29–32. [Online]. Available: https://doi.org/10.1109/CSIE.2009.275

[16] W. Wang, "Helper function inlining in dynamic binary translation," in *CC '21: 30th ACM SIGPLAN International Conference on Compiler Construction, Virtual Event, Republic of Korea, March 2-3, 2021*, A. Smith, D. Demange, and R. Gupta, Eds. ACM, 2021, pp. 107–118. [Online]. Available: https://doi.org/10.1145/3446804.3446851

[17] T. Spink, H. Wagstaff, and B. Franke, "A retargetable system-level DBT hypervisor," in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, D. Malkhi and D. Tsafrir, Eds. USENIX Association, 2019, pp. 505–520. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/spink

[18] V. J. Reddi, D. Connors, R. Cohn, and M. D. Smith, "Persistent code caching: Exploiting code reuse across executions and applications," in *Fifth International Symposium on Code Generation and Optimization (CGO 2007), 11-14 March 2007, San Jose, California, USA*. IEEE Computer Society, 2007, pp. 74–88. [Online]. Available: https://doi.org/10.1109/CGO.2007.29

[19] W. Wang, P. Yew, A. Zhai, and S. McCamant, "A general persistent code caching framework for dynamic binary translation (DBT)," in *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, A. Gulati and H. Weatherspoon, Eds. USENIX Association, 2016, pp. 591–603. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/wang

[20] A. D'Antras, C. Gorgovan, J. D. Garside, and M. Luján, "Optimizing indirect branches in dynamic binary translators," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 1, pp. 7:1–7:25, 2016. [Online]. Available: https://doi.org/10.1145/2866573

[21] N. Jia, C. Yang, Y. He, and X. Cheng, "DTT: program structure-aware indirect branch optimization via direct-tpc-table in DBT system," in *Computing Frontiers Conference, CF'14, Cagliari, Italy - May 20 - 22, 2014*, P. Trancoso, D. Franklin, and S. A. McKee, Eds. ACM, 2014, pp. 12:1–12:10. [Online]. Available: https://doi.org/10.1145/2597917.2597944

[22] N. Jia, C. Yang, J. Wang, D. Tong, and K. Wang, "SPIRE: improving dynamic binary translation through spc-indexed indirect branch redirecting," in *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (co-located with ASPLOS 2013), VEE '13, Houston, TX, USA, March 16-17, 2013*, S. Muir, G. Heiser, and S. M. Blackburn, Eds. ACM, 2013, pp. 1–12. [Online]. Available: https://doi.org/10.1145/2451512.2451516

[23] X. Zhang, X. Gao, Q. Guo, J. Huang, H. Liu, and X. Meng, "VBIW: optimizing indirect branch in dynamic binary translation," in *10th IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, HPCC/EUC 2013, Zhangjiajie, China, November 13-15, 2013*. IEEE, 2013, pp. 1456–1462. [Online]. Available: https://doi.org/10.1109/HPCC.and.EUC.2013.206

[24] S. Fu, J. Wu, and W. Hsu, "Improving SIMD code generation in QEMU," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, W. Nebel and D. Atienza, Eds. ACM, 2015, pp. 1233–1236. [Online]. Available: http://dl.acm.org/citation.cfm?id=2757098

[25] Y. Liu, D. Hong, J. Wu, S. Fu, and W. Hsu, "Exploiting asymmetric SIMD register configurations in arm-to-x86 dynamic

binary translation," in *26th International Conference on Parallel Architectures and Compilation Techniques, PACT 2017, Portland, OR, USA, September 9-13, 2017*. IEEE Computer Society, 2017, pp. 343–355. [Online]. Available: https://doi.org/10.1109/PACT.2017.15

[26] S. Fu, D. Hong, Y. Liu, J. Wu, and W. Hsu, "Efficient and retargetable SIMD translation in a dynamic binary translator," *Softw. Pract. Exp.*, vol. 48, no. 6, pp. 1312–1330, 2018. [Online]. Available: https://doi.org/10.1002/spe.2573

[27] ——, "Optimizing data permutations in structured loads/stores translation and SIMD register mapping for a cross-isa dynamic binary translator," *J. Syst. Archit.*, vol. 98, pp. 173–190, 2019. [Online]. Available: https://doi.org/10.1016/j.sysarc.2019.07.008

[28] M. Kristien, T. Spink, B. Campbell, S. Sarkar, I. Stark, B. Franke, I. Böhm, and N. P. Topham, "Fast and correct load-link/store-conditional instruction handling in DBT systems," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 39, no. 11, pp. 3544–3554, 2020. [Online]. Available: https://doi.org/10.1109/TCAD.2020.3013048

[29] Z. Zhao, Z. Jiang, Y. Chen, X. Gong, W. Wang, and P. Yew, "Enhancing atomic instruction emulation for cross-isa dynamic binary translation," in *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, J. W. Lee, M. L. Soffa, and A. Zaks, Eds. IEEE, 2021, pp. 351–362. [Online]. Available: https://doi.org/10.1109/CGO51591.2021.9370312

[30] R. A. Sokolov and A. V. Ermolovich, "Background optimization in full system binary translation," *Program. Comput. Softw.*, vol. 38, no. 3, pp. 119–126, 2012. [Online]. Available: https://doi.org/10.1134/S0361768812030073

[31] E. G. Cota and L. P. Carloni, "Cross-isa machine instrumentation using fast and scalable dynamic binary translation," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2019, Providence, RI, USA, April 14, 2019*, J. B. Sartor, M. Naik, and C. J. Rossbach, Eds. ACM, 2019, pp. 74–87. [Online]. Available: https://doi.org/10.1145/3313808.3313811

[32] P. Kedia and S. Bansal, "Fast dynamic binary translation for the kernel," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, M. Kaminsky and M. Dahlin, Eds. ACM, 2013, pp. 101–115. [Online]. Available: https://doi.org/10.1145/2517349.2522718

[33] S. Rokicki, E. Rohou, and S. Derrien, "Hardware-accelerated dynamic binary translation," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, D. Atienza and G. D. Natale, Eds. IEEE, 2017, pp. 1062–1067. [Online]. Available: https://doi.org/10.23919/DATE.2017.7927147

[34] T. Spink, H. Wagstaff, and B. Franke, "Hardware-accelerated cross-architecture full-system virtualization," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, pp. 36:1–36:25, 2016. [Online]. Available: https://doi.org/10.1145/2996798