Effective Exploitation of SIMD Resources in Cross-ISA Virtualization

Jin Wu
Harbin Institute of Technology, China

Jian Dong
Harbin Institute of Technology, China

Ruili Fang
University of Georgia
USA

Ziyi Zhao
Nankai University
China

Xiaoli Gong
Nankai University
China

Wenwen Wang
University of Georgia
USA

Decheng Zuo
Harbin Institute of Technology, China

Abstract

System virtualization is a fundamental technology that enables many important applications. However, existing virtualization techniques suffer from a critical limitation due to their limited exploitation of host SIMD hardware resources, especially when a guest application *does not* have inherently fine-grained data-level parallelism. To bridge this utilization gap and unleash the full potential of host SIMD resources, this paper proposes an *effective* and *unconventional* SIMD exploitation technique. The proposed exploitation takes advantage of ample host SIMD registers and powerful host SIMD instructions to generate more efficient host binary code for guest applications even without any fine-grained data-level parallelism. It also mitigates the shortage of general-purpose registers on the host platform, as well as improves the efficiency of accessing guest registers. We have implemented the exploitation in an extensively-used virtualization platform, QEMU. Experimental results on a comprehensive list of benchmarks from PARSEC, SPEC-CPU2017, and Google Octane JavaScript benchmark suite show that an average of 2.2X performance speedup can be achieved for AArch64 binaries on an x86-64 host machine. We believe the proposed technique will provide a new perspective for our community to rethink the exploitation of SIMD hardware resources.

CCS Concepts: • **Software and its engineering → Virtual machines**; **Just-in-time compilers**; **Dynamic compilers**.

Keywords: SIMD optimization, Cross-ISA virtualization, Dynamic binary translation, QEMU

VEE '21, April 16, 2021, Virtual, USA

ACM ISBN 978-1-4503-8394-3/21/04…$15.00
https://doi.org/10.1145/3453933.3454016

ACM Reference Format:
Jin Wu, Jian Dong, Ruili Fang, Ziyi Zhao, Xiaoli Gong, Wenwen Wang, and Decheng Zuo. 2021. Effective Exploitation of SIMD Resources in Cross-ISA Virtualization. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '21), April 16, 2021, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3453933.3454016

1 Introduction

System virtualization plays a fundamental role in many important applications, such as supporting software/hardware product transitions [7, 25], workload migration/consolidation across heterogeneous ISAs [8], mobile computation offloading [44], mobile application development [2], and mobile gaming [9, 46]. However, existing virtualization systems suffer from a significant limitation due to the limited exploitation of hardware resources for cross-ISA virtualization.

As an innovative execution model to exploit fine-grained data-level parallelism, single instruction multiple data (SIMD) has been implemented and integrated into modern CPUs in the form of hardware extensions, e.g., Intel SSE/AVX/AVX-512, ARM NEON and PowerPC AltiVec. Moreover, extensive research efforts have been devoted to leveraging these SIMD extensions to address problems in a variety of application domains, including but not limited to, algorithms [16, 17], databases [20, 30, 49], compiler optimizations [3, 21], language runtimes [11, 37], and deep learning [1, 13]. Nevertheless, further investigations on existing SIMD-related optimizations for cross-ISA virtualization [12, 22, 23, 29] reveal that almost *all* of these optimizations attempt to leverage SIMD extensions to exploit the data-level parallelism in virtualized applications. This implies that applications that have very few or even no data-level parallelism opportunities will not benefit from these hardware SIMD extensions.

TABLE 1 illustrates the storage resources offered by several popular SIMD extensions. Here we assume the CPU processors have 64-bit architectures, i.e., x86-64 and AArch64. As shown in the table, compared to general-purpose registers, all of the SIMD extensions can provide *more* storage capacity via SIMD registers. To further understand how these SIMD registers are used, we then measure the dynamic percentage of SIMD instructions in total executed instructions in several

Table 1. Register resources provided by SIMD extensions

	SSE	AVX	AVX-512	NEON
Number of SIMD Reg	16	16	32	32
Bit width of SIMD Reg	128	256	512	128
SIMD Reg Capacity	256B	512B	2KB	512B
General Reg Capacity	128B	128B	128B	256B

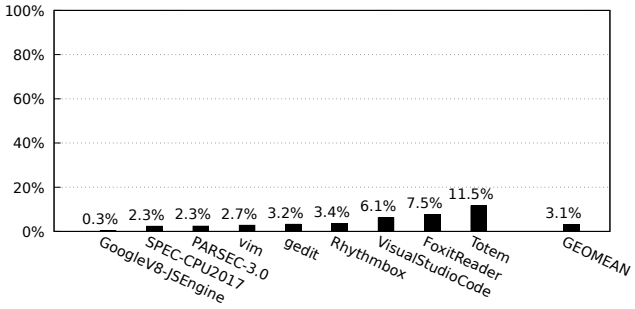

Figure 1. Dynamic ratio of SIMD instructions. All applications are compiled by GCC-7.4.0 with SIMD enabled and tested on an x86-64 CPU with AVX-512.

representative applications, given that SIMD registers can only be accessed through SIMD instructions. The result is shown in Figure 1. Note that all applications are compiled with the most aggressive SIMD optimizations available in the compilers. Overall, SIMD instructions account for only an average of 3% dynamic instructions. This also leads to significant under-utilization of SIMD extensions in cross-ISA virtualization systems for these applications.

To overcome the above limitations and unleash the full potential of hardware SIMD extensions, this paper blazes a trail to *effectively* and *unconventionally* exploit SIMD extensions in cross-ISA virtualization. In particular, we consider SIMD registers as "regular" registers and utilize them in a way similar to regular general-purpose registers. This is inspired by the observation that most SIMD extensions support not only vector operations but also *scalar* operations. This way, we can greatly increase the register storage capacity, as well as produce more optimization opportunities by leveraging the powerful functionalities provided by SIMD instructions. We believe the proposed exploitation technique will provide a new perspective for our community to rethink the current way to exploit hardware SIMD extensions.

The core technology that enables cross-ISA virtualization is dynamic binary translation (DBT), which translates binary code from a *guest* ISA to a different *host* ISA at runtime. By executing the generated host binary code, DBT can emulate the functionality of the guest binary code on a host physical machine. One of the key technical challenges of DBT is how to emulate guest general-purpose registers, especially when the guest architecture has *more* registers than the host architecture. A popular solution for this problem is to use host memory locations and translate guest register accesses into host memory accesses. Obviously, this approach incurs substantial performance overhead due to the frequent memory accesses and additional memory load and store instructions. In contrast, our SIMD exploitation overcomes this challenge elegantly by leveraging host SIMD registers to emulate guest general-purpose registers. Moreover, this emulation scheme enables the cross-ISA virtualization system to further utilize host powerful SIMD instructions to generate more efficient and scalable host binary code.

We have implemented the proposed SIMD exploitation technique into a prototype based on a widely-used virtualization platform, QEMU [4]. Our prototype supports both AArch64 and RISCV64 guest ISAs and x86-64 host ISA. To evaluate the effectiveness of the prototype, we use a broad range of benchmarks from various benchmark suites, including PARSEC [5], SPEC-CPU2017 [36], and Google Octane JavaScript benchmark suite [14]. Experimental results show that our SIMD exploitation technique can achieve significant performance speedup, e.g., an average of 2.2X performance speedup can be achieved for PARSEC benchmarks with AArch64 binaries on an x86-64 host machine. Moreover, the performance overhead introduced by exploiting SIMD resources is negligible.

In summary, this paper makes the following contributions:

- We propose an unconventional exploitation of SIMD hardware resources in cross-ISA virtualization. The exploitation does not rely on any data-level parallelism. To the best of our knowledge, this is the first attempt to exploit SIMD resources in this new perspective.
- We implement the proposed exploitation of SIMD extensions in a research prototype based on QEMU, which is a real-world and widely-used virtualization system. We also address several implementation challenges in order to achieve better performance efficiency.
- We conduct comprehensive experiments to evaluate the proposed exploitation using a wide range of benchmarks and applications. Experimental results demonstrate that significant performance speedup can be attained with the proposed SIMD exploitation.

The rest of this paper is organized as follows. Section 2 presents the background knowledge and the motivation. Section 3 describes the proposed unconventional exploitation of SIMD extensions in DBT. Section 4 explains how to implement the proposed exploitation in a real-world virtualization system. Section 5 shows the experimental results. Section 6 discusses related work. And Section 7 concludes the paper.

2 Background and Motivation

In this section, we present a brief introduction to SIMD and DBT, as well as motivate our SIMD exploitation for cross-ISA virtualization based on DBT.

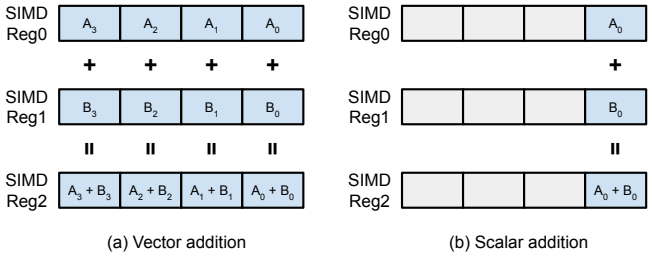

Figure 2. Hardware SIMD extensions support both vector and scalar operations.

2.1 SIMD

The SIMD execution model was invented to exploit fine-grained data-level parallelism for better performance efficiency and scalability. By simultaneously conducting the same computations on multiple input data, the SIMD model can produce multiple results at the same time. Besides, the time cost of a SIMD operation is very similar to that of a normal or scalar operation, which, however, can only produce a single computation result at a time.

To embrace the powerful SIMD model, processor manufacturers have brought it into modern commercial CPUs through hardware extensions. Notable examples include Intel SSE/AVX/AVX-512 and ARM NEON. In general, a hardware SIMD extension consists of an array of SIMD registers and a set of SIMD instructions, which are used to manipulate the SIMD registers and exported to software through different ISA extensions. The major difference between a SIMD register and a general-purpose register is that the data stored in a SIMD register can be packed as a *vector* to participate in vector operations specified by SIMD instructions.

2.1.1 Flexible Operation Types. It is worth noting that most SIMD extensions support not only vector operations but also *scalar* operations. Figure 2 shows the major difference between these two operation types. In particular, a scalar operation only takes as input the least significant bits of the source SIMD registers, e.g., the lowest 64 bits, and updates the corresponding least significant bits of the destination SIMD register. This flexibility enables us to use SIMD registers as general-purpose registers because we only need to maintain the least significant bits.

2.1.2 Ample SIMD Registers. As we can see in TABLE 1, SIMD registers in all listed extensions offer much more storage capacity than regular general-purpose registers. For instance, sixteen additional SIMD registers were introduced when Intel AVX-512 was launched. On the other side, the quantity and the bit width of general-purpose registers typically remain the same for architectural compatibility. An example is the x86-64 ISA, which has only sixteen 64-bit general-purpose registers. As a result, by exploiting the plentiful SIMD registers, even without identifying the data-level parallelism, we can reduce the pressure on the limited general-purpose registers and potentially achieve considerable performance improvements.

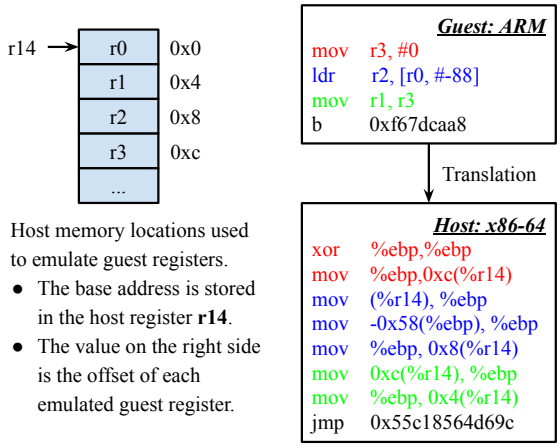

Figure 3. Emulation of guest registers using host memory locations in existing cross-ISA virtualization systems.

2.1.3 Powerful SIMD Instructions. Most SIMD instructions, both vector and scalar instructions, are typically more powerful than corresponding regular instructions. This is mainly embodied in two aspects. First, SIMD instructions can encode more complicated operations than regular instructions. For example, in some SIMD extensions, a SIMD instruction can accomplish a multiplication along with an addition. Second, SIMD instructions are capable of taking more operands than regular instructions. For example, an AVX-512 instruction has the capability to manipulate three or four operands, while most regular x86-64 instructions can only have two operands. These advanced features produce more opportunities to generate optimized binary code with SIMD instructions.

2.2 DBT

DBT is one of the key technologies that power cross-ISA virtualization. It dynamically translates guest binary code into *semantically-equivalent* host binary code. Previous research work has demonstrated that, in most cases, more than 90% of the execution time of a DBT system is consumed by executing the translated host binary code [33, 42]. Therefore, it is crucial for a DBT system to generate efficient host binary code for better performance. The efficiency of guest register emulation takes an important portion of the overall performance of a DTB system.

Motivation. To design a practical DBT system, a fundamental technical impediment is *how to emulate guest general-purpose registers*. This is of great importance because general-purpose registers are a key component of an ISA and accessed

very frequently by various instructions. A straightforward solution is simply mapping each guest register into one host register and using this dedicated host register to emulate the guest register. However, this approach cannot work when the host ISA has *less* general-purpose registers than the guest ISA. For example, AArch64 and x86-64 have 32 and 16 general-purpose registers, respectively. In fact, even if the guest and the host ISAs have the same quantity of general-purpose registers, it is still quite challenging to adopt this approach. The reason is that it is sometimes required to reserve some host general-purpose registers for temporary usage in order to emulate the functionality of guest instructions, due to the semantic gaps between the guest and host ISAs.

Therefore, most existing DBT systems such as QEMU, address the above limitation by exploiting host memory to emulate guest general-purpose registers. Figure 3 shows an example of this mechanism. Here, the guest ISA is ARM while the host ISA is x86-64. For example, the host memory location at `r14 + 0x4` represents the emulated ARM `r1` register. With this mechanism, a DBT system can decouple the emulation of guest registers from the number of available host registers and emulate as many guest registers as necessary.

However, this memory-based register emulation suffers heavy performance overhead due to the large amount of the translated host instructions and massive memory access. The right side of Figure 3 shows an example. We use the same color to indicate the original ARM instruction and the corresponding x86-64 instructions translated from the ARM instruction using this emulation mechanism. As shown in the figure, translated code is two times larger than the original ARM binary code in terms of number of instructions. Hence, it is imperative to invent a new mechanism to emulate guest registers for better performance. To this end, we propose to exploit host SIMD registers for efficient guest register emulation. Details will be presented in the next section.

3 Exploiting SIMD in DBT

There are several basic principles to design an optimization approach for a DBT system.

- **Generality.** It is desired if the optimization can be applied to general scenarios rather than just some specific cases. This determines whether the approach is able to explore extensive optimization opportunities and cover a broad variety of application scopes.
- **Adaptability.** Given the remarkable differences between different architectures and ISAs, the approach is expected to be able to adapt to various execution environments. This allows the approach to take full advantage of guest and host features to achieve maximum performance improvement.

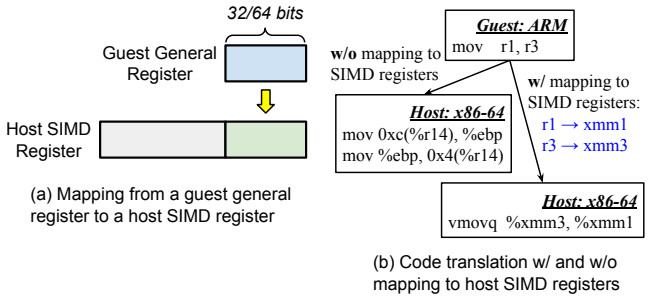

Figure 4. Exploiting host SIMD registers to emulate guest general-purpose registers in cross-ISA virtualization.

- **Independence.** The optimization approach should not make any assumption on the implementation details of the DBT system. Otherwise, it cannot work if the DBT system is not implemented as expected.

It is worth noting that the exploitation of SIMD extensions proposed in this paper satisfies the above principles. Specifically, it provides a general solution to take advantage of the SIMD extension on the host machine to enhance the emulation efficiency of a DBT system.

3.1 Mapping Guest General-Purpose Registers to Host SIMD Registers

As discussed before, it incurs heavy performance overhead if guest general-purpose registers are emulated by host memory locations. To mitigate this overhead, we propose to leverage host SIMD registers to emulate guest general-purpose registers. Considering that the bit width of a SIMD register is typically larger than that of a general-purpose register even if they are from different ISAs, only the *least significant bits* of host SIMD registers are used for register emulation. Therefore, such mapping mechanism is as shown in Figure 4(a). The example in Figure 4(b) illustrates the host instructions translated from the guest instruction with and without mapping guest general-purpose registers to host SIMD registers. It shows that the host memory accesses are eliminated by the mapping, and a host SIMD scalar instruction is generated to emulate the functionality of the guest instruction.

Though the emulation scheme is intuitive, there are several technical challenges to put it into practice. We next describe these challenges and our solutions in detail.

3.1.1 Resolving Host SIMD Registers Used by Code Translator. The first obstacle is that the host SIMD registers can also be used by the translator. Typically, the translator is written in high-level programming languages, such as C/C++, and compiled by a native compiler, e.g., GCC. It is foreseeable that SIMD instructions may be generated during the compilation of the translator. As a result, the SIMD registers accessed by these SIMD instructions can pose a *conflict* if they are also used for guest register emulation.

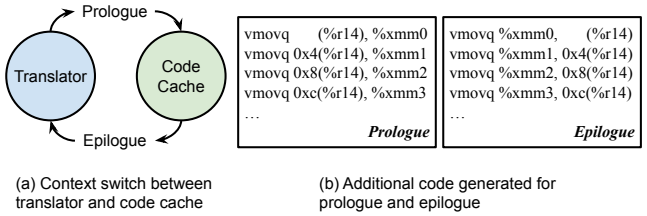

Figure 5. Exploiting host SIMD registers to emulate guest general-purpose registers in cross-ISA virtualization.

To address this issue, we extend the *prologue* and *epilogue* to resolve the potential conflict. In general, there are two execution environments in a DBT system. One is the translator and the other is the translated host binary code or *code cache*. Since these two environments share the same host machine in a time-sharing manner, a *context switch* is required when the execution is transferred between each other. This work is realized by executing the code placed in prologue and epilogue, as depicted in Figure 5(a). In our design, we restore/save the host SIMD registers from/to the corresponding host memory locations in the prologue and epilogue, respectively. Figure 5(b) shows an example of the additional code appended to the prologue and epilogue. To further reduce the performance overhead introduced by saving/restoring SIMD registers, our approach employs an offline analysis to detect the potential conflicts in advance. This allows us to only save SIMD registers that are actually used by the translator. This way, we can resolve the aforementioned conflict.

3.1.2 Resolving Host SIMD Registers Used in Translated Host Binary Code. Apart from the translator, host instructions generated by the translator may use host SIMD registers as well, e.g., to emulate guest SIMD registers. This can also lead to conflicts if a guest general-purpose register is mapped to the same host SIMD register. Therefore, we also need to resolve the potential conflict.

To this end, we conduct a comprehensive study on a variety of open-source DBT systems. This study aims to uncover whether host SIMD registers are used in the generated host binary code and if yes, how they are used. Our study covers not only popular DBT systems but also common dynamic binary instrumentation systems because they are also built with DBT techniques. We manually examine the source code of each system to understand how it works and investigate the SIMD-related implementation details. Some DBT systems support multiple host ISAs, we then check all host ISAs with SIMD extensions. To validate the conclusions, we also develop some guest programs to test the systems. The details are omitted here due to space limitations.

Table 2 illustrates the results of the study. As shown in the table, for some DBT systems, e.g., QEMU with x86-64 host, the host SIMD registers are used in the generated host binary code only when guest applications contain SIMD instructions. But, for other DBT systems, host SIMD registers are not used at all because guest SIMD applications are not supported yet. These results suggest that most host SIMD registers are rarely used in the translated host binary code, especially when guest applications have very few SIMD instructions. Thus, there exists an opportunity to exploit such resources to emulate guest general-purpose registers.

Table 2. SIMD registers used by host binary code generated in popular open-source DBT systems. "G" means host SIMD registers are used only when guest code uses guest SIMD registers; "N" means host SIMD registers are not used in translated host code; "-" means the corresponding host is not supported.

	x86-64	AArch64	RISCV64	PowerPC
QEMU [4]	G	N	N	G
Dyninst [26]	G	-	-	-
Dolphin [9]	G	G	-	G
Valgrind [27]	G	G	-	G
DynamoRIO [6]	G	G	-	-
AndroidEmu [2]	G	-	-	-

Inspired by the results of the above study, we resolve the potential conflicts when exploiting host SIMD registers by providing a programmable interface for the target DBT system. This interface allows the DBT system to specify which host SIMD register is used in the generated host binary code when translating guest binary code. The interface is defined as the following two routines:

```
void mark_host_simd_reg_begin(int regno);
void mark_host_simd_reg_end(int regno);
```

They are expected to be invoked at the start and end points of the usage of the specified host SIMD register, respectively. Besides, they need to be invoked inside a basic block. Each time when the *begin* routine is invoked, we firstly check whether the specified host SIMD register is used for guest register emulation. If yes, additional host binary code will be generated to spill the emulated guest register from the host SIMD register to the corresponding host memory location so that the host SIMD register can be vacated. Otherwise, nothing will happen. Correspondingly, when the *end* routine is invoked, we restore the emulated guest register from the host memory location to the host SIMD register. Note that it may involve multiple instructions to access the same SIMD register between the *begin* and *end* routines. In this way, we can remove the conflicts caused by the utilization of SIMD registers in the translated host binary code.

3.1.3 Which Guest Registers Should be Mapped? The final challenge we face is that the host architecture may not have sufficient SIMD registers to emulate *all* guest general-purpose registers. This is possible if the host is equipped with a less powerful or old generation of the SIMD extension. So

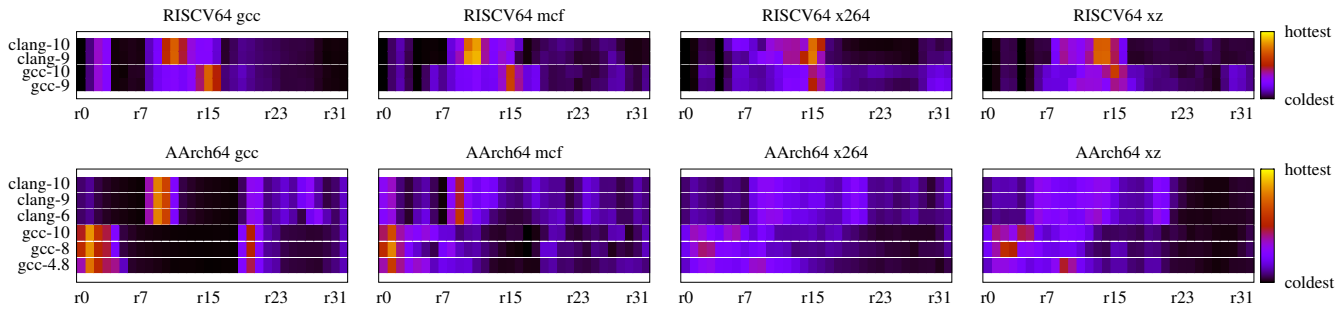

Figure 6. Hotness of registers across different applications and different compilers. Here the hotness of a register is defined as the percentage of occurrences of this register in dynamically-executed instructions.

the problem here is which guest registers should be selected to be emulated using the limited host SIMD registers.

To overcome this challenge, we propose to map guest registers based on the *hotness* of the guest registers. A potential concern is that general-purpose registers may experience different hotness in different applications. To address this concern, we measure the hotness of general-purpose registers in applications on different architectures. Figure 6 shows the results. We only present some applications with representative hotness patterns. As shown in the figure, the hotness of the registers are quite steady across different applications compiled by the same compiler with the same/different versions. This shows the practicability of the proposed hotness-based selection scheme. From Figure 6, we can also see differences across different compilers. The potential reason is that different compilers may employ different register allocation algorithms. Therefore, our hotness-based selection scheme generates a specific mapping profile for each popular compiler.

3.2 Generating Optimized Host Code with SIMD Instructions

With guest general-purpose registers mapped to host SIMD registers, we can employ host SIMD instructions to manipulate the emulated guest registers. This provides us opportunities to generate more optimized host binary code given that SIMD instructions often have more advanced and powerful features than regular instructions. In our exploitation of SIMD extensions, we leverage features that are available on most popular SIMD extensions. We next describe them in detail, as well as how to exploit them in DBT systems.

3.2.1 More Operands. The quantity of the operands that an instruction can encode reflects the data-processing capability of this instruction. In general, regular instructions in some popular ISAs, e.g., x86-64, can operate only two operands. But, regular instructions in other ISAs, e.g., AArch64, can have three operands. The inconsistency between different ISAs in the quantity of operands inevitably lead to poor

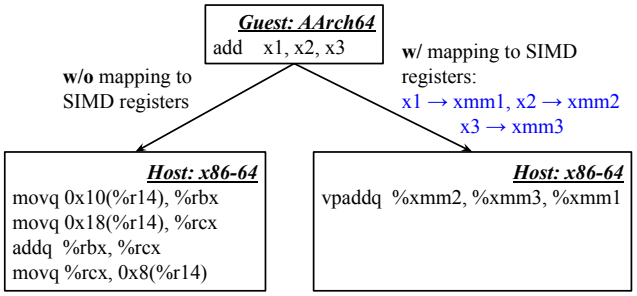

Figure 7. Exploiting host SIMD operands to generate efficient host code.

efficiency in the translated host binary code. For example, when translating from an AArch64 guest to an x86-64 host, the translator has to generate more than one x86-64 instruction to incorporate the operands in an AArch64 instruction.

Fortunately, SIMD instructions are often capable of taking more operands than regular instructions. For example, the x86-64 AVX instructions can take three operands. As a result, by leveraging host SIMD instructions, we can generate more efficient host binary code. The example in Fig. 7 shows the host instructions translated from the guest instruction with and without using the host SIMD instructions. As shown in the example, to emulate the guest instruction, four regular host instructions are generated. Among the four instructions, the later two instructions are generated because each x86-64 instruction can only have two operands. In contrast, by mapping guest registers to host SIMD registers and leveraging host SIMD instructions, only one host instruction is required to emulate the guest instruction.

To achieve this, we need to select host SIMD instructions for binary code translation. But, this requires that the involved guest general-purpose registers are emulated by host SIMD registers. Therefore, we create a table to record the current emulation status of each guest general-purpose register. When a guest instruction is translated, we look up this table to check whether the registers involved in this instruction

Algorithm 1: Host Instruction Selection for Translation.

Input: T - Table that indicates guest register emulation status
I - Guest instruction to be translated
Output: SIMD/REGULAR - Instruction selection decision

```
1  flag ← true;
2  simd_operand ← 0;
3  r_transfer ← null;
4  foreach guest register r accessed by I do
5      if Query(T, I, r) = host SIMD register then
6          simd_operand ← simd_operand + 1;
7      else
8          r_transfer ← r;
9      end
10 end
11 if simd_operand ≥ 2 then
12     if r_transfer ≠ null then
13         r_tmp ← Alloc_tmp_SIMD_register();
14         Gen_host_mov(r_tmp ← r_transfer);
15         Replace r_transfer in I with r_tmp
16     end
17     flag ← true;
18 else
19     flag ← false;
20 end
21 if flag then
22     return SIMD;
23 else
24     return REGULAR;
25 end
```

are emulated using host SIMD registers. If yes, appropriate host SIMD instructions can be selected to emulate the guest instruction. Otherwise, regular host instructions will be generated. **Algorithm 1** shows the details about how to make the decision of host instruction selection in the translator. It simply counts if more than 2 operands out of 3 are XMM registers. If so, the translator takes the SIMD instruction generation branch. Here note that it may be necessary to generate several complementary regular instructions in some cases even if the SIMD policy is selected, due to the restrictions enforced by the host ISA on SIMD registers. An example is that SIMD registers are typically not allowed to be used in memory operands, and thus additional regular instructions are required to move the memory address in a host SIMD register to a temporary host general-purpose register in order to perform a memory access. The selection algorithm can be enhanced by multiple possible solutions, such as comprehensive context analysis or even machine learning technology, which is out of the scope of this paper.

3.2.2 More Powerful Opcodes.

In most ISAs, especially RISC ISAs, an opcode of a regular instruction can only encode one operation, e.g., a memory access or an arithmetic/logic operation. In contrast, a SIMD opcode can perform the same operation on *multiple* data. The side effect of this feature is similar to performing *multiple same* operations on different data at the same time. By exploiting this feature, we can potentially generate more efficient host binary code by replacing several host instructions, which are translated from multiple guest instructions accordingly, with a single SIMD instruction. Figure 8 illustrates such an example. As shown in the figure, without emulating guest general-purpose registers using host SIMD registers, eight host instructions are generated for the five gust instructions. But, if host SIMD registers are employed to emulate guest registers, we can exploit powerful SIMD instructions and only three SIMD instructions are required to emulate the guest instructions.

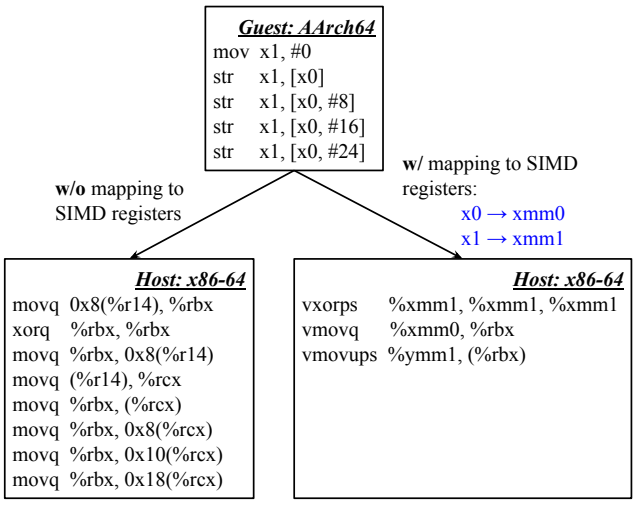

Figure 8. Exploiting host SIMD opcodes to generate efficient host code.

To leverage the powerful SIMD opcodes, we need to identify multiple guest instructions that can be emulated by host SIMD instructions. There are two key points: (1) the identified guest instructions should perform the same operation; (2) the operation is supported by host SIMD instructions. To this end, when a basic block is translated, we analyze guest instructions in this block. Specifically, we check whether any of the operations performed by multiple guest instructions can be emulated by a single host SIMD instruction.

4 Implementation

We have implemented the proposed exploitation of SIMD extensions in a prototype based on QEMU [4] (version 4.2.0), which is a cross-ISA virtualization platform and has been widely used in many research projects and industry products. Our current implementation supports both AArch64 and RISCV64 as the guest ISAs and x86-64 as the host ISA.

QEMU employs the tiny code generator (TCG) to translate guest binary code to host binary code. TCG is a *retargetable* translator, which means it supports multiple guest ISAs and multiple host ISAs. Typically, it requires significant engineering efforts to achieve retargetability in a DBT system, because of the complexity and diversity of different ISAs. As a result, TCG introduces an intermediate representation (IR) to mitigate the engineering efforts. Each operation in the IR is represented by a TCG Op, which consists of an opcode and multiple operands. Hence, the guest binary code is firstly translated to TCG IRs and then host binary code. This way, TCG is able to decouple the implementation details related to guest and host ISAs and mitigate the engineering efforts required to support multiple guest and host ISAs.

Next, we describe the implementation issues we encountered when integrating the proposed exploitation of SIMD extensions into QEMU and our solutions for these issues.

4.1 Cooperating with TCG

TCG emulates guest general-purpose registers using host memory locations. The consistency between the emulated guest registers and the host memory locations is maintained at either the end of an instruction translation or the boundaries of basic blocks. That is, the host memory locations are updated at the end of each block to keep the latest values of the emulated guest registers. To integrate the proposed exploitation of host SIMD extensions, our implementation extends TCG to emulate guest general-purpose registers using host SIMD registers. More specifically, we create a new operand type for TCG Ops: `SIMD Reg`. If a guest register is emulated by a host SIMD register, it is firstly translated into a TCG operand with the `SIMD Reg` type in TCG Ops and then mapped to a host SIMD register when TCG Ops are translated to host binary code. Besides, the update to the host memory location that corresponds to this guest register is removed at the ends of blocks. For the guest registers that are not emulated by host SIMD registers, we still use the original mechanism in TCG to emulate them.

In addition, we further extend TCG to generate host SIMD instructions. This is realized by introducing new TCG opcodes, which are designed to manipulate TCG operands with the `SIMD Reg` type. TCG Ops of these new opcodes can be translated to host SIMD instructions by TCG backends.

4.2 Supporting Guest Conditional Codes

Condition codes summarize the execution results of the current instruction and can be used in the following instructions. For example, AArch64 has four condition codes: NF, ZF, CF, and VF. TCG emulates guest condition codes in a way similar to guest general-purpose registers, i.e., using host memory locations. The emulated condition codes are updated based on the semantics of the guest instructions that define the condition codes. Since we emulate guest registers using host SIMD registers, we firstly leverage host SIMD instructions to calculate the results of guest condition codes. In case host SIMD instructions produce different condition code results compared to general instructions, e.g., x86-64 `test` and `ptest` as shown in Table 3, or no host SIMD instruction is available, we generate host instructions to move emulated values from host SIMD registers to host general-purpose registers and then conduct the calculations.

Table 3. Different condition codes modified by `test` and `ptest` instructions

	Negative	Zero	Positive
test %rax,%rax	SF IF	PF ZF IF	PF IF
ptest %xmm1, %xmm1	CF IF	CF ZF IF	CF IF

4.3 Handling Helper Functions

TCG relies on *helper functions* to emulate complicated and obscure guest instructions. These functions are typically developed in high-level programming languages for simplicity and expected to be invoked directly from code cache through host *call* instructions. Note that the helper functions are executed in the translator context, which is different from the code cache context. This poses an implementation issue of our exploitation of host SIMD registers to emulate guest general-purpose registers. That is, the values in host SIMD registers can be corrupted by a helper function call if the helper function also accesses the host SIMD registers.

To solve this issue, a straightforward approach is to perform context switches before and after a helper function is invoked from code cache. However, this solution would introduce extremely high performance overhead because helper functions are invoked frequently.

To address this issue, we conduct a thorough study on *all* helper functions in QEMU, including their source code and binary code. Our study reveals several interesting findings. First, most helper functions use none or very few host SIMD registers. Second, a small amount of helper functions dominate a large part of dynamic helper function calls. For example, *helper_lookup_tb_ptr* accounts for 85.55% to 99.99% of all helper function calls. Finally, the source code of helper functions is quite stable, as they are used to emulate guest instructions, which typically do not change. These findings inspire us to develop an offline analysis of helper functions to figure out which host SIMD register(s) is/are used in each helper function. With this information, we then extend TCG to generate customized host binary code to save and restore SIMD registers on demand for different helper functions. This allows us to mitigate the performance overhead incurred by invoking helper functions.

5 Experimental Results

To evaluate the effectiveness of the proposed exploitation of SIMD extensions, we conduct experiments on a wide range

Table 4. Configurations of our evaluation platform

	Configuration	
CPU	Intel Xeon W-2145 at 3.70GHz	
	# of Cores	8
	# of Threads	16
	Private L1d	32KB
	Private L1i	32KB
	Private L2	1024KB
	Shared L3	11MB
	SIMD Extensions	SSE, AVX, AVX-512
Memory	64GB	
Operating System	Ubuntu 18.04.4 with Linux-4.15	

of benchmark applications, including PARSEC [5] (version 3.0), SPEC-CPU2017 [36], Octane JavaScript applications [14] (version 2.0). These applications cover various problem domains.

5.1 Experimental Setup

5.1.1 Benchmarks. The PARSEC benchmark suite was originally designed to study the performance of shared-memory multi-threaded programs on chip-multiprocessors. It has been widely used in many research projects to evaluate the multi-thread performance of a computer system. The suite includes applications in computer vision, content search, physics simulation, video encoding, and data mining.

SPEC-CPU2017 is an industry-standardized CPU-intensive benchmark suite. It has been widely used to measure and compare computer systems. There are two types of benchmarks in this suite: integer (or *int* for short) and floating point (or *fp* for short). Our evaluation uses the *speed* configuration, which adopts the execution time as the performance metric. The dataset used in this section is *ref*. Under this configuration, SPEC-CPU2017 benchmarks can be divided into two sets: *intspeed* and *fpspeed*. Besides, OpenMP threads are available for all fpspeed benchmarks, while only one intspeed benchmark, i.e., *xz*, can use OpenMP threads. For these benchmarks, we test them with maximum hardware threads. In addition, *cam4* in fpspeed is excluded from the evaluation due to the crash in the original QEMU.

Given that JavaScript is one of the most popular programming languages for web applications, Our evaluation also includes JavaScript applications from the *Octane* benchmark suite. To run these JavaScript applications, we use a real-world JavaScript engine, i.e., Google V8 [15]. It is also the JavaScript engine used in the Google Chrome web browser.

In addition to the default compilation options, we also enable the auto-vectorization optimizations to deliberately generate more guest SIMD instructions in the binaries. However, there are NEON instructions in AArch64 binaries but no vector instructions in RISCV64 binaries. Besides, QEMU does not support vector registers for RISCV64 guests. Since PARSEC and Octane were not designed for RISCV, a small part of test applications were failed with compilation or at run time, which are excluded from the experimental results.

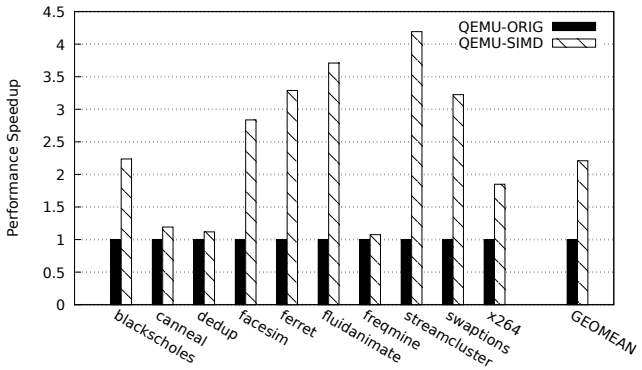

Figure 9. Performance speedup achieved by our SIMD exploitation with AArch64 as the guest ISA for PARSEC benchmarks with 16 threads. The original QEMU is the performance baseline.

5.1.2 Evaluation Platform. Table 4 shows the detailed configurations of our evaluation platform, which is exclusively occupied during the evaluation. The program execution characteristic is analyzed for understanding the optimized DBT system better. For performance evaluation, to remove the potential noises caused by random factors, we evaluate each benchmark application three times, and take the arithmetic mean of the three runs as the final performance result of this application.

5.2 Performance Results

Experiments are carried out to evaluate the performance with SIMD exploitation. Results on both AArch64 and RISCV64 guest platforms show a noticeable improvement compared to the original emulation.

5.2.1 AArch64. Figure 9 shows the performance speedup achieved for PARSEC benchmarks. Here, the number of threads is 16 and the performance baseline is the original QEMU without the proposed SIMD exploitation. As shown in the figure, we can achieve performance improvement for all evaluated benchmarks. For some benchmarks, e.g., *streamcluster*, the performance speedup can be as high as 4.19X. On average, our SIMD exploitation achieves 2.2X speedup. This demonstrates the capability of our SIMD exploitation to optimize the performance of cross-ISA virtualization.

Figure 10 shows the performance speedup achieved by our SIMD exploitation for SPEC-CPU2017 *intspeed* benchmarks. Here, the performance baseline is the original QEMU without SIMD exploitation. As shown in the figure, we can achieve performance speedup for all *intspeed* benchmarks. An interesting observation is that the performance speedup for *xz* is significantly higher than other applications. This is

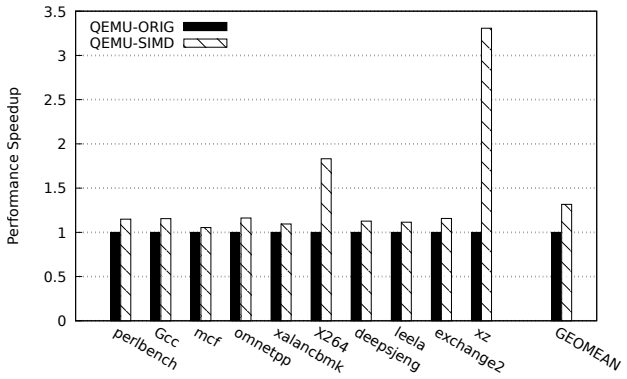

Figure 10. Performance speedup achieved by our SIMD exploitation for SPEC-CPU2017 *intspeed* benchmarks with AArch64 as guest. The baseline is the original QEMU. All benchmarks are evaluated with one thread except *xz*, which has 16 threads.

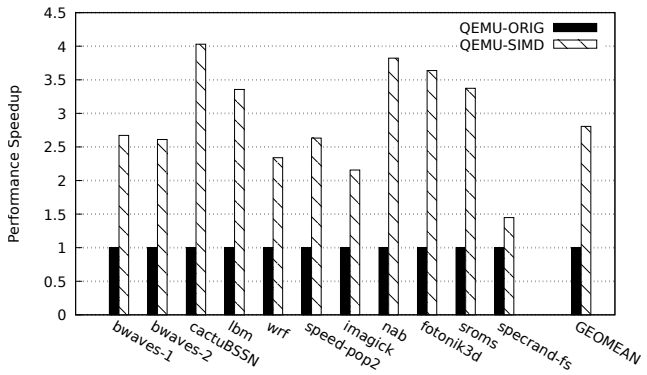

Figure 11. Performance speedup achieved by our SIMD exploitation for SPEC-CPU2017 *fpspeed* benchmarks with AArch64 as guest. The baseline is the original QEMU. All benchmarks are evaluated with 16 threads.

because *xz* has multiple threads, while other applications are single-threaded. To emulate multiple guest threads, QEMU needs to maintain the guest register status for each thread. This inevitably increases the pressure on the host memory system, as the last level cache is shared between different processor cores. In contrast, by mapping guest registers to host SIMD registers, we can mitigate the cache pressure by taking advantage of SIMD registers available on each physical core. On average, our SIMD exploitation can achieve 1.38X performance speedup for SPEC-CPU2017 *intspeed* benchmarks. This demonstrates the effectiveness of our SIMD exploitation to optimize QEMU by utilizing SIMD hardware resources.

Similarly, Figure 11 illustrates the performance speedup achieved by the SIMD exploitation for SPEC-CPU2017 *fpspeed* benchmarks compiled to AArch64. As shown in the

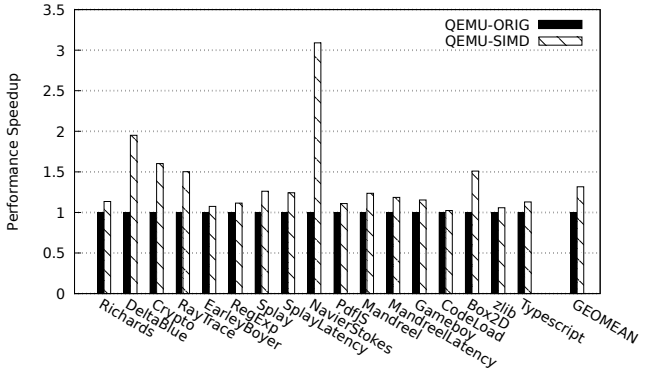

Figure 12. Performance speedup achieved by our SIMD exploitation for Octane JavaScript applications running on the Google V8 JavaScript engine with AArch64 as guest. The baseline is the original QEMU.

figure, with our SIMD exploitation, QEMU attains performance improvements for all evaluated benchmarks. For some benchmarks, such as *cactuBSSN*, the performance speedup can be as high as 4X. Overall, an average of 2.8X speedup can be observed for *fpspeed* benchmarks. This is higher than the average speedup for *intspeed* benchmarks. The reason is that, as mentioned before, all *fpspeed* benchmarks can be compiled with OpenMP threads and achieve more performance benefits from the proposed SIMD exploitation.

Figure 12 shows the performance speedup achieved by our SIMD exploitation for Octane JavaScript applications running on the Google V8 JavaScript engine. The performance baseline is the original QEMU without our exploitation. As shown in the figure, we can achieve performance improvement for all evaluated JavaScript applications. For some applications, for example, *NavierStokes*, the performance speedup is as high as 3X. On average, our SIMD exploitation obtains 1.32X performance speedup. This demonstrates the effectiveness of our SIMD exploitation on real-world applications.

5.2.2 RISCV64. Similar experiments are conducted for RISCV64 guest applications. Our SIMD exploitation shows performance improvement as well. As illustrated in Fig. 13 for PARSEC applications with 16 threads, our approach achieves up to 11.64% speedup with an average of 6.5%. Fig 14 shows the performance improvement of SPEC-CPU2017 *intspeed*. *Exchange2* achieves the highest speedup which is 15.45%. The geometric mean of all the benchmarks is 7.37%. The experimental results of Google V8 JavaScript engine with Octane as test applications are shown in Fig. 15. The geometric mean of speedup is 15.31%.

5.2.3 Results Analysis. We further analysed the generated binary of SPEC-CPU2017 with SIMD exploitation, taking intspeed as testing applications with ref input and RISCV-64 as the guest platform. We measured the static and dynamic

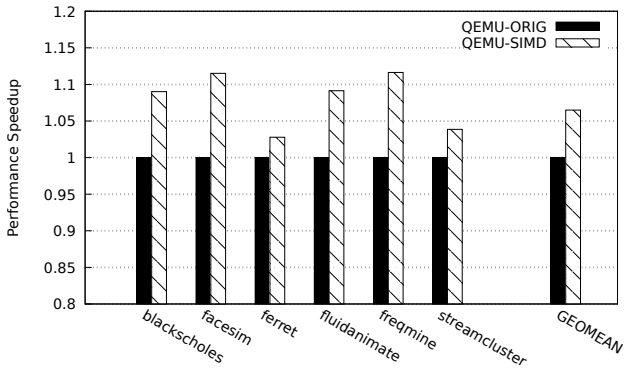

Figure 13. Performance speedup achieved by our SIMD exploitation with RISCV64 as guest for PARSEC benchmarks with 16 threads. The baseline is the original QEMU.

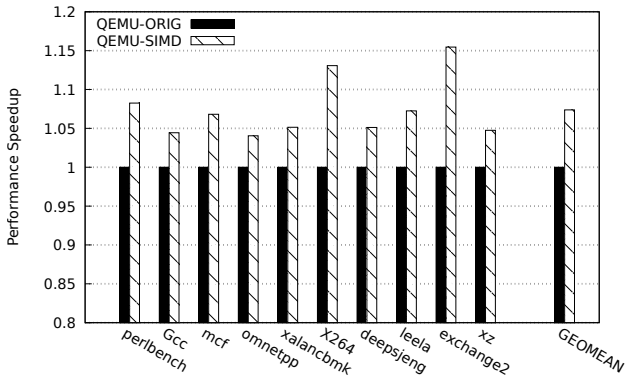

Figure 14. Performance speedup achieved by our SIMD exploitation for SPEC-CPU2017 *intspeed* with RISCV64 as guest. The baseline is the original QEMU. All benchmarks are evaluated with one thread except *xz*, which has 16 threads.

characteristics of the optimized DBT system, in order to understand the improvement brought by SIMD exploitation.

Memory access and SIMD instructions in the host code are counted. On average, memory access takes 39.26% of all the instructions generated by the original translator. While with SIMD exploitation, it is reduced to 31.48%. SIMD instructions, which take nearly 0% in the original translator generated code, is increased to 14.53% with the SIMD enhanced approach. The quantity of instructions in the code cache is reduced by 6% on average. At run time, the executed instructions are 5.15% less than the original QEMU. Details are illustrated in Table 5. For the second test of *xz*, the instruction reduction ratio is much higher than others. Further study shows its cache miss rate is significantly higher than others, i.e., as high as 30%, which indicates a large number of loads and stores with very poor data locality were performed. With the SIMD exploitation, frequent data movement between

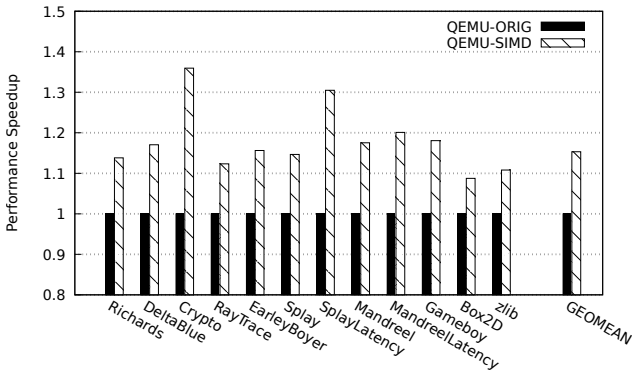

Figure 15. Performance speedup achieved by our SIMD exploitation for Octane JavaScript applications running on the Google V8 JavaScript engine with RISCV64 as guest. The baseline is the original QEMU.

Table 5. Instruction statistics on SPEC-CPU2017 *intspeed* with RISCV64 as guest. "Mem" means memory access instructions, "SIMD" means SIMD instructions, "w/o Opt" means original QEMU, "w/ Opt" means QEMU with SIMD exploitation, "Rdc-D" means dynamically reduced instructions, "Rdc-S" means statically reduced instructions.

	Mem w/o Opt	Mem w/ Opt	SIMD w/o Opt	SIMD w/ Opt	Rdc-D	Rdc-S
perbench-1	37.54%	30.11%	0.00%	12.94%	4.20%	3.81%
perbench-2	37.68%	30.36%	0.00%	13.02%	4.28%	3.74%
perbench-3	37.73%	30.20%	0.00%	12.75%	4.31%	4.08%
gcc-1	37.68%	30.75%	0.00%	11.64%	4.21%	4.45%
gcc-2	37.76%	30.83%	0.00%	11.71%	4.45%	4.50%
gcc-3	37.68%	30.75%	0.00%	11.65%	4.43%	4.45%
mcf	38.36%	31.44%	0.00%	13.51%	3.68%	4.26%
omnetpp	41.36%	35.94%	0.00%	11.66%	4.55%	4.64%
xalancbmk	42.79%	36.73%	0.00%	13.40%	6.51%	7.17%
x264-1	39.35%	31.99%	0.00%	18.57%	3.46%	2.48%
x264-2	40.18%	31.46%	0.00%	19.48%	3.00%	3.38%
x264-3	40.53%	31.33%	0.00%	19.68%	4.45%	4.04%
deepsjeng	38.24%	28.93%	0.00%	14.93%	4.93%	4.81%
leela	40.87%	32.70%	0.00%	15.10%	5.82%	5.71%
exchange2	39.80%	30.14%	0.00%	15.83%	6.85%	6.79%
xz-1	39.94%	30.53%	0.00%	15.74%	5.54%	5.54%
xz-2	39.92%	31.04%	0.00%	15.46%	12.89%	28.13%
Average	39.26%	31.48%	0.00%	14.53%	5.15%	6.00%

memory emulated register and host register can be significantly mitigated. Since the SIMD register mapped guest registers won't consume cache resources anymore, the cache pressure is also reduced because of less memory access.

AArch64 architecture attains additional benefit with the combination of existing guest SIMD instructions and our exploitation. Different from RISCV64, AArch64 binary contains SIMD instructions, which are translated to host SIMD instructions or corresponding helper functions at run time. So, there is an opportunity to perform more SIMD instructions within host SIMD registers, without additional data moving. As a result, AArch64 applications experience higher performance improvement than RISCV64 applications.

5.3 Translation Overhead

Finally, we study the translation overhead introduced by the proposed SIMD exploitation. To this end, we use the hardware performance monitor unit to measure the CPU cycles consumed by the translation process with and without our SIMD exploitation. On average, the CPU cycles consumed by translating each benchmark application with the SIMD exploitation is in the range of 2.03E+10 to 1.59E+12. This corresponds to an average of 1.45% performance slow down of the entire translation process. We believe this translation overhead is negligible for a cross-ISA DBT system.

6 Related Work

6.1 SIMD Optimizations

A considerable amount of optimizations have been proposed to exploit SIMD extensions available on commercial CPU processors [1, 11, 13, 16, 17, 20, 30, 37, 49]. These SIMD optimizations often target different application domains. For instance, it has been a long history to utilize SIMD extensions in database systems [20, 30, 49]. Zhou et al. leverage SIMD instructions to implement common database operations, including sequential scans, aggregation, index operations, and joins [49]. Polychroniou et al. adapt analytical databases to take advantage of more recent advances of SIMD extensions, such as wider SIMD registers and more advanced SIMD instructions [30]. Besides, SIMD extensions have been adopted to optimize graph and sorting algorithms [16, 17]. Moreover, language runtimes have integrated SIMD support for enhanced efficiency [11, 37].

In addition, a variety of auto-vectorization mechanisms have been proposed to automatically identify fine-grained data-level parallelism and generate SIMD instructions for general sequential programs [3, 18, 21, 28, 31, 32]. In order to generate SIMD instructions, these mechanisms typically employ program analysis techniques and sophisticated vectorization algorithms to recognize data-level parallelism and resolve data dependence during the compilation process. For example, SuperGraph-SLP constructs a large code region, called SuperGraph, to eliminate inaccuracies in the vectorization process by providing a holistic view of the code [31].

A common point of these SIMD-related optimizations exploit SIMD extensions by merely exploring the fine-grained data-level parallelism in the target applications. This is different from our exploitation of SIMD extensions, because our exploitation does not rely on data-level parallelism in target applications. Instead, we take advantage of hardware resources offered by SIMD extensions to achieve performance improvement for applications with very few or even no data-level parallelism opportunities. We believe our exploitation would complement existing SIMD-related optimizations to unleash the full potential of hardware SIMD extensions.

6.2 Optimizing DBT with SIMD Exploitation

Given the critical role of DBT in cross-ISA virtualization, some optimizations focus on SIMD extensions [12, 23, 24, 29]. A noticeable example is saSLP [23], which lifts guest SIMD code to LLVM IR and then performs vectorization on LLVM IR to exploit higher degrees of parallelism available on host SIMD extensions. It is worth pointing out that the key requirement of applying these DBT optimizations is that the guest binary code is already vectorized. Although some of the scalar instructions can be vectorized by the aforementioned techniques, a large amount of regular guest instructions can not fit such restricted requirements. That implies it is hard for guest binary code with very few or even no SIMD instructions to benefit from these optimizations. In contrast, our exploitation is not limited to vectorized guest code, because we utilize the hardware resources offered by host SIMD extensions to enhance DBT systems for a wide range of guest applications.

There are also some other DBT-related optimizations [10, 19, 34, 35, 38–41, 43, 45, 47, 48]. In general, it is possible for the SIMD exploitation proposed in this paper to collaborate with them to further enhance the efficiency of DBT systems.

7 Conclusion

SIMD extensions have become a permanent and important hardware component in modern CPU processors. Existing SIMD-related optimizations in cross-ISA virtualization exploit SIMD extensions by only exploring fine-grained data-level parallelism in guest applications. This leads to quite poor utilization of the hardware resources provided by hardware SIMD extensions. To address this issue, this paper proposes an effective and unconventional exploitation of SIMD extensions in cross-ISA virtualization, which does not rely on data-level parallelism in guest applications. Specifically, the exploitation takes advantage of the hardware resources of SIMD extensions, including ample SIMD registers and powerful SIMD instructions, to generate more efficient host binary code. We also implement a prototype of the exploitation based on a practical cross-ISA virtualization platform, QEMU. Experimental results on a wide range of benchmark applications demonstrate that the proposed exploitation is able to achieve significant performance improvements.

Acknowledgments

We are very grateful to the anonymous reviewers for their valuable feedback and comments. This work is partially supported by the Natural Science Foundation of Tianjin, China (18JCYBJC15600), the National Key Research and Development Program of China (2018YFB1003405). This work is supported in part by a faculty startup funding of the University of Georgia.

References

[1] Berkin Akin, Zeshan A. Chishti, and Alaa R. Alameldeen. 2019. ZCOMP: Reducing DNN Cross-Layer Memory Footprint Using Vector Extensions. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 126–138. https://doi.org/10.1145/3352460.3358305

[2] Android. 2020. Run apps on the Android Emulator. https://developer.android.com/studio/run/emulator.

[3] Sara S. Baghsorkhi, Nalini Vasudevan, and Youfeng Wu. 2016. FlexVec: Auto-Vectorization for Irregular Loops. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 697–710. https://doi.org/10.1145/2908080.2908111

[4] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(USENIX ATC '05)*. USENIX Association, USA, 41.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. Association for Computing Machinery, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[6] Derek L. Bruening and Saman Amarasinghe. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. Dissertation. Massachusetts Institute of Technology, USA.

[7] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 2003. The Transmeta Code MorphingTM Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (San Francisco, California, USA) *(CGO '03)*. IEEE Computer Society, USA, 15–24.

[8] Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. 2012. Execution Migration in a Heterogeneous-ISA Chip Multiprocessor. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 261–272. https://doi.org/10.1145/2150976.2151004

[9] Dolphin Emulator Project. 2020. A GameCube and Wii emulator. https://dolphin-emu.org.

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

[11] Carol Eidt and Tanner Gooding. 2020. SIMD Support in .NET: Abstract and Concrete Vector Types and Operations. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 229–241. https://doi.org/10.1145/3368826.3377926

[12] Sheng-Yu Fu, Ding-Yong Hong, Yu-Ping Liu, Jan-Jan Wu, and Wei-Chung Hsu. 2017. 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* (Barcelona, Spain) *(LCTES 2017)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/3078633.3081029

[13] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. 2018. Anatomy of High-Performance Deep Learning Convolutions on SIMD Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Article 66, 12 pages.

[14] Google. 2020. The JavaScript Benchmark Suite for the modern web. https://developers.google.com/octane.

[15] Google. 2020. V8 JavaScript engine. https://v8.dev.

[16] Shuo Han, Lei Zou, and Jeffrey Xu Yu. 2018. Speeding Up Set Intersections in Graph Algorithms Using SIMD Instructions. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1587–1602. https://doi.org/10.1145/3183713.3196924

[17] Kaixi Hou, Hao Wang, and Wu-chun Feng. 2015. ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on X86-Based Many-Core Processors. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 383–392. https://doi.org/10.1145/2751205.2751247

[18] Joonmoo Huh and James Tuck. 2017. Improving the Effectiveness of Searching for Isomorphic Chains in Superword Level Parallelism. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 718–729. https://doi.org/10.1145/3123939.3124554

[19] Jinhu Jiang, Rongchao Dong, Zhongjun Zhou, Changheng Song, Wenwen Wang, Pen-Chung Yew, and Weihua Zhang. 2020. More with Less – Deriving More Translation Rules with Less Training Data for DBTs Using Parameterization. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 415–426. https://doi.org/10.1109/MICRO50266.2020.00043

[20] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13 (Sept. 2018), 2209–2222. https://doi.org/10.14778/3275366.3284966

[21] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. 2013. When Polyhedral Transformations Meet SIMD Code Generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 127–138. https://doi.org/10.1145/2491956.2462187

[22] Jianhui Li, Qi Zhang, Shu Xu, and Bo Huang. 2006. Optimizing Dynamic Binary Translation for SIMD Instructions. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '06)*. IEEE Computer Society, USA, 269–280. https://doi.org/10.1109/CGO.2006.27

[23] Y. Liu, D. Hong, J. Wu, S. Fu, and W. Hsu. 2017. Exploiting Asymmetric SIMD Register Configurations in ARM-to-x86 Dynamic Binary Translation. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 343–355.

[24] Charith Mendis, Ajay Jain, Paras Jain, and Saman Amarasinghe. 2019. Revec: Program Rejuvenation through Revectorization. In *Proceedings of the 28th International Conference on Compiler Construction* (Washington, DC, USA) *(CC 2019)*. Association for Computing Machinery, New York, NY, USA, 29–41. https://doi.org/10.1145/3302516.3307357

[25] Microsoft. 2018. How x86 emulation works on ARM. https://docs.microsoft.com/en-us/windows/uwp/porting/apps-on-arm-x86-emulation.

[26] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. 1995. The Paradyn Parallel Performance Measurement Tool. *Computer* 28, 11 (Nov. 1995), 37–46. https://doi.org/10.1109/2.471178

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

[28] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. 2011. Vapor SIMD: Auto-Vectorize Once, Run Everywhere. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, USA, 151–160.

[29] Yihan Pang, Robert Lyerly, and Binoy Ravindran. 2019. Cross-ISA Execution of SIMD Regions for Improved Performance. In *Proceedings of the 12th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '19)*. Association for Computing Machinery, New York, NY, USA, 55–67. https://doi.org/10.1145/3319647.3325832

[30] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1493–1508. https://doi.org/10.1145/2723372.2747645

[31] V. Porpodas. 2017. SuperGraph-SLP Auto-Vectorization. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 330–342.

[32] Vasileios Porpodas, Rodrigo C. O. Rocha, Evgueni Brevnov, Luís F. W. Góes, and Timothy Mattson. 2019. Super-Node SLP: Optimized Vectorization for Code Sequences Containing Operators and Their Inverse Elements. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 206–216.

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

[34] Changheng Song, Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Weihua Zhang. 2019. Unleashing the Power of Learning: An Enhanced Learning-Based Approach for Dynamic Binary Translation. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 77–89.

[35] Tom Spink, Harry Wagstaff, and Björn Franke. 2019. A Retargetable System-Level DBT Hypervisor. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 505–520.

[36] Standard Performance Evaluation Corporation. 2020. SPEC CPU 2017. https://www.spec.org/cpu2017.

[37] Alen Stojanov, Ivaylo Toskov, Tiark Rompf, and Markus Püschel. 2018. SIMD Intrinsics on Managed Language Runtimes. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 2–15. https://doi.org/10.1145/3168810

[38] Wenwen Wang. 2021. Helper Function Inlining in Dynamic Binary Translation. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction* (Virtual, Republic of Korea) *(CC 2021)*. Association for Computing Machinery, New York, NY, USA, 107–118. https://doi.org/10.1145/3446804.3446851

[39] Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. 2018. 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* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 84–97. https://doi.org/10.1145/3173162.3177160

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

[41] Wenwen Wang, Jiacheng Wu, Xiaoli Gong, Tao Li, and Pen-Chung Yew. 2018. Improving Dynamically-Generated Code Performance on Dynamic Binary Translators. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Williamsburg, VA, USA) *(VEE '18)*. Association for Computing Machinery, New York, NY, USA, 17–30. https://doi.org/10.1145/3186411.3186413

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

[43] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. 2020. Efficient and Scalable Cross-ISA Virtualization of Hardware Transactional Memory. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 107–120. https://doi.org/10.1145/3368826.3377919

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

[45] Jin Wu, Jian Dong, Ruili Fang, Wenwen Wang, and Decheng Zuo. 2020. PerfDBT: Efficient Performance Regression Testing of Dynamic Binary Translation. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*. 389–392. https://doi.org/10.1109/ICCD50377.2020.00071

[46] Qifan Yang, Zhenhua Li, Yunhao Liu, Hai Long, Yuanchao Huang, Jiaming He, Tianyin Xu, and Ennan Zhai. 2019. Mobile Gaming on Personal Computers with Direct Android Emulation. In *The 25th Annual International Conference on Mobile Computing and Networking* (Los Cabos, Mexico) *(MobiCom '19)*. Association for Computing Machinery, New York, NY, USA, Article 19, 15 pages. https://doi.org/10.1145/3300061.3300122

[47] Ziyi Zhao, Zhang Jiang, Ying Chen, Xiaoli Gong, Wenwen Wang, and Pen-Chung Yew. 2021. Enhancing Atomic Instruction Emulation for Cross-ISA Dynamic Binary Translation. In *19th IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2021)*. Association for Computing Machinery, New York, NY, USA.

[48] Ziyi Zhao, Zhang Jiang, Ximing Liu, Xiaoli Gong, Wenwen Wang, and Pen-Chung Yew. 2020. DQEMU: A Scalable Emulator with Retargetable DBT on Distributed Platforms. In *49th International Conference on Parallel Processing - ICPP* (Edmonton, AB, Canada) *(ICPP '20)*. Association for Computing Machinery, New York, NY, USA, Article 7, 11 pages. https://doi.org/10.1145/3404397.3404403

[49] Jingren Zhou and Kenneth A. Ross. 2002. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/564691.564709