Effective and Unconventional Exploitation of SIMD Extensions in Cross-ISA Virtualization

***Abstract*—As a promising hardware component on commercial CPU processors, SIMD extensions have been broadly exploited in various problem domains. However, most of existing SIMD-related optimizations exploit SIMD extensions by merely exploring the inherent and fine-grained data-level parallelism in target applications, and therefore cannot work for applications with very few or even no data-level parallelism opportunities. This results in extremely poor utilization of the hardware resources offered by prevalent SIMD extensions. To bridge this utilization gap and unleash the full potential of hardware SIMD extensions, this paper proposes an effective and unconventional exploitation of SIMD extensions. We choose cross-ISA virtualization to demonstrate our exploitation, as it plays a fundamental role in a wide range of important areas. Our exploitation takes advantage of the hardware resources provided by SIMD extensions, including ample SIMD registers and powerful SIMD instructions, to generate more efficient binary code for cross-ISA virtualization systems. We have implemented our exploitation in an extensively-used cross-ISA virtualization platform, QEMU. Experimental results on a comprehensive list of benchmarks from PARSEC, SPEC-CPU2017, and JavaScript applications demonstrate that significant performance improvements can be achieved by our exploitation, e.g., an average of 2.2X performance speedup is achieved for PARSEC benchmarks. We believe the exploitation proposed in this paper provides a new perspective to rethink the current way to exploit hardware SIMD extensions.**

I. Introduction

Single instruction multiple data (SIMD) is an innovative execution model to exploit fine-grained data-level parallelism. It has been implemented and integrated into modern commercial CPU processors in the form of hardware extensions, e.g., Intel SSE/AVX/AVX-512, ARM NEON, RISC-V RVV, 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 [1], [2], databases [3]–[5], compiler optimizations [6], [7], language runtimes [8], [9], and deep learning [10], [11].

To program a SIMD extension, the typical process includes the following three steps: (1) identifying the data-level parallelism in the problem to be solved, (2) mapping such parallelism to SIMD registers, and (3) employing SIMD instructions to perform parallel computations on SIMD registers. Given the inherent complexity and the significant engineering efforts required in these steps, it is extremely difficult, if not impossible, to program SIMD extensions from scratch without the loss of parallelism opportunities, especially for non-expert programmers. Therefore, a considerable amount of programming models and compiler tools have been created to facilitate SIMD programming, e.g., the Intel ISPC compiler [12], LLVM vectorizers [13], and the pragma "**#pragma** omp simd" in OpenMP 4.0 [14]. These tools substantially alleviate the programming burden of SIMD extensions, as well as delivering observable performance and scalability improvements by utilizing SIMD extensions.

TABLE I
Register Storage resources provided by SIMD extensions

	SSE	AVX	AVX-512	NEON
Number of SIMD Registers	16	16	32	32
Bit width of SIMD Registers	128	256	512	128
SIMD Register Capacity	256B	512B	2KB	512B
Gen.-Purp. Register Capacity	128B	128B	128B	256B

Further investigations on existing SIMD-related optimizations reveal that almost *all* of these optimizations attempt to leverage SIMD extensions to exploit the data-level parallelism in target applications. On the surface, this is not surprising, as indeed the SIMD extensions were introduced for such a purpose. However, on the other side, this also implies that the applications that have very few or even no data-level parallelism opportunities cannot benefit from these hardware SIMD extensions. Even worse, no matter a SIMD extension is used or not, it occupies some processor dies and thus consumes more or less energy. Therefore, if a processor is mainly used to run applications without data-level parallelism, the SIMD extension of the processor becomes a huge waste and potentially increases the cost of computing.

TABLE I 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. For AVX-512, the storage capacity of SIMD registers is even as high as 2 KB. To further understand how these SIMD registers are used, we then measure the dynamic percentage of SIMD instructions in total executed instructions in several representative applications, given that SIMD registers can only be accessed through SIMD instructions. These applications cover a wide range of categories, such as industry-standard benchmark suite, text editor, development tool, PDF reader, media player, and JavaScript engine. The result is shown in Fig. 1. We aggregate the results of benchmarks in SPEC-CPU2017 and PARSEC-3.0 together due to the space limitation. Note that all applications are compiled with the most aggressive SIMD optimizations available in the compilers. As shown in the figure, the percentage of SIMD instructions is surprisingly low in all evaluated applications.

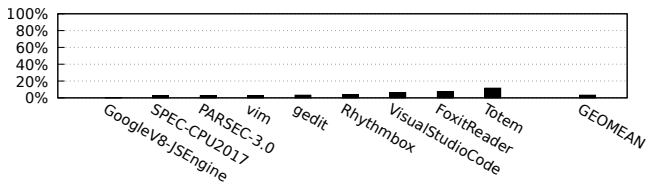

Fig. 1. Dynamic ratio of SIMD instructions. All applications are compiled by GCC-7.4.0 with SIMD enabled and tested on an x86-64 CPU with AVX-512.

Overall, SIMD instructions account for only an avearge of 3% dynamic instructions. This demonstrates that SIMD extensions are significantly underutilized in these applications. In fact, the major reason behind this phenomenon is that it is very hard to identify fine-grained data-level parallelism in these applications, even with mature compiler tools.

To bridge the above utilization gap and unleash the full potential of hardware SIMD extensions, this paper blazes a trail to *effectively* and *unconventionally* exploit SIMD extensions in cross-ISA (i.e., instruction set architecture) 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. More importantly, our exploitation does not rely on data-level parallelism, which enables us to take advantage of the resources offered by SIMD extensions, including SIMD registers and SIMD instructions, even in applications that have no data-level parallelism. To the best of our knowledge, this is the first attempt to extend the usage scenarios of SIMD extensions to cover the applications that have no data-level parallelism. We believe this research will provide a new perspective for our community to rethink the current way to exploit hardware SIMD extensions.

Cross-ISA virtualization plays a fundamental role in many important applications, such as supporting software/hardware product transitions [15], [16], workload migration/consolidation across heterogeneous ISAs [17], mobile computation offloading [18], mobile application development [19], and mobile gaming [20], [21]. 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 is able to emulate the functionality of the guest binary code on a host physical machine.

One of the key technical challenges to design a DBT system is how to emulate guest general-purpose registers, especially when the guest architecture has more registers than the host architecture. A straightforward solution is mapping guest registers to host memory locations and translating guest register accesses to host memory accesses. Obviously, this approach can incur substantial performance overhead due to the frequent memory accesses. Therefore, to address this challenge, we propose to unconventionally exploit the SIMD extension available on the host machine. Specially, we leverage host SIMD registers to emulate guest general-purpose registers. This can not only remove many memory access instructions in the generated host binary code, but also produce potential optimization opportunities because many computations now can be performed directly on SIMD registers. We have implemented a prototype atop a widely-used virtualization platform, QEMU [22]. Experimental results on a wide range of benchmarks and real-world applications show that our SIMD exploitation can achieve significant performance speedup. For example, an average of 2.2X performance speedup can be achieved for PARSEC benchmarks.

In summary, this paper makes the following contributions:

- We propose an unconventional exploitation of hardware SIMD extensions in cross-ISA virtualization. Different from existing SIMD-related optimizations, our exploitation does not depend on any data-level parallelism. To the best of our knowledge, this is the first attempt to exploit SIMD extensions 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. The evaluation covers a wide range of benchmarks and real-world 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 II presents the background knowledge and the motivation. Section III describes the proposed unconventional exploitation of SIMD extensions in DBT. Section IV explains how to implement the proposed exploitation in a real-world virtualization system. Section V shows the experimental results. Section VI discusses related work. And Section VII concludes the paper.

II. 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.

A. SIMD

The SIMD execution model was invented to exploit fine-grained data-level parallelism for better performance efficiency and scalability. Essentially, the SIMD model is a simple form of parallel processing. 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 operation, which, however, can only produce a single computation result. Therefore, the SIMD model can boost program performance significantly. In fact, its idea and design philosophy have been lent to create today's widely-deployed

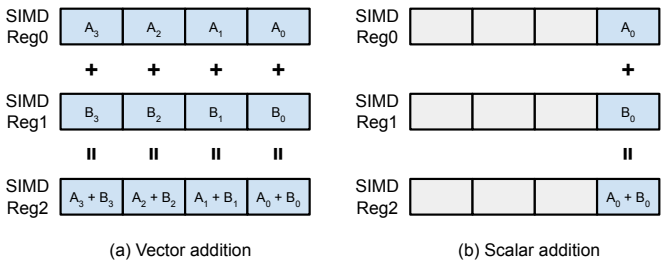

Fig. 2. Hardware SIMD extensions support both vector and scalar operations.

graphics processing unit (GPU) processors and many other domain-specific accelerators [23]–[25].

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. The length of the vector depends on the type of the data elements in the vector. For instance, each AVX-512 register is 512-bit and thus can hold up to 8 double precision or 16 single precision floating-point values. In addition, different generations of the SIMD extension on the same architecture are typically backward compatible, e.g., AVX-512 to AVX and SSE.

1) Flexible Operation Types: It is worth noting that most SIMD extensions support not only vector operations but also *scalar* operations. Fig. 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. Furthermore, when memory operands are involved, we can use a scalar operation to manipulate less memory data, which otherwise cannot be achieved by a vector operation due to the wide vector length. Thus, we build our exploitation of SIMD extensions based on this flexibility.

2) Ample SIMD Registers: As shown in TABLE I, SIMD registers in all extensions offer more storage capacity than regular general-purpose registers. This is reasonable, as SIMD registers generally need to hold more input and output data than general-purpose registers. Besides, a newer generation of a SIMD extension is often strengthened with either wider SIMD registers or more SIMD registers. For instance, sixteen additional SIMD registers were introduced when AVX-512 was launched to replace AVX. 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 thus achieve desired performance improvements.

3) Powerful SIMD Instructions: Another important advantage of SIMD extensions is that 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 produces more opportunities to generate optimized binary code with SIMD instructions. Therefore, it is possible to achieve better performance by leveraging SIMD extensions but without depending on the data-level parallelism.

B. 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. The unit of the translation is a *basic block*, or *block* for short. Each block consists of a sequence of guest instructions with only one entry and one exit. That is, a block can contain at most one branch instruction at the end of the block. The translation result, i.e., the generated host binary code, is saved to a software-managed *code cache* to mitigate the translation overhead, as one block may be executed multiple times in the same execution run. After a block is translated, the generated host binary code is executed to emulate the functionality of the guest binary code on the host physical machine. 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 [26], [27]. Therefore, it is crucial for a cross-ISA virtualization system to generate efficient host binary code for better performance.

1) Emulation of Guest Registers: To design a practical DBT system, a critical 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

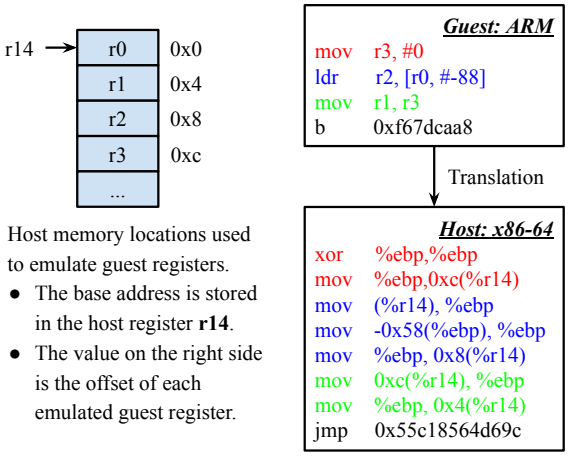

Fig. 3. Emulation of guest registers in cross-ISA virtualization.

order to emulate the functionality of guest instructions, due to the semantic gaps between the guest and host ISAs.

Therefore, most of existing DBT systems, such as QEMU, address the above limitation by exploiting host memory to emulate guest general-purpose registers. Specifically, each guest register is mapped to a host memory location with the same bit width, and the accesses to a guest register are then translated to the accesses to the corresponding host memory location. Fig. 3 shows an example of this emulation mechanism. Here, the guest ISA is ARM while the host ISA is x86-64. The starting address of the host memory locations is stored in the host register `r14` to facilitate the accesses to the emulated guest registers. 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 mechanism suffers heavy performance overhead due to the large amount of the translated host instructions. The right side of Fig. 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, this mechanism results in substantial bloating in the translated x86-64 binary code, which is two times larger than the original ARM binary code in term 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. More details will be presented in the next section.

III. Exploiting SIMD in DBT

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

- **General.** It is desired if the optimization approach 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.
- **Adaptable.** Given the remarkable differences between different architectures and ISAs, the approach is expected to be able to adapt to various environments. This allows the approach to take full advantage of guest and host features to achieve maximum performance improvement.
- **Independent.** The optimization approach should not make any assumption on the DBT system. In particular, the approach should not rely on any specific 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. It takes advantage of the SIMD extension available on the host machine to enhance the emulation efficiency of a DBT system.

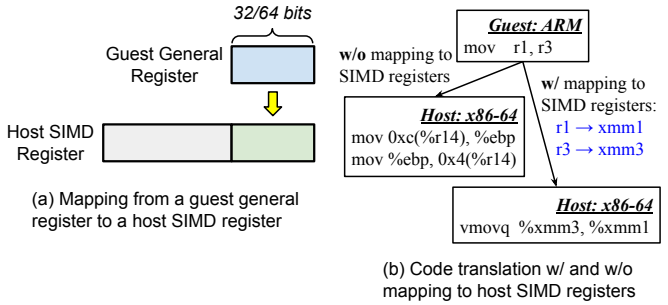

Fig. 4. Exploiting host SIMD registers to emulate guest general-purpose registers in cross-ISA virtualization.

A. Mapping Guest 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 performance 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, we choose to use the *least significant bits* of host SIMD registers for register emulation. This design choice aligns with common scalar operations supported by SIMD extensions because most of these operations only manipulate these bits of SIMD registers. Therefore, a guest general-purpose register is mapped to a host SIMD register, as shown in Fig. 4(a). The example in Fig. 4(b) illustrates the host instructions translated from the guest instruction with and without mapping guest general-purpose registers to host SIMD registers. It demonstrates that the host memory access instructions are eliminated by mapping guest registers to host SIMD registers, and a host SIMD scalar instruction is generated to emulate the functionality of the guest instruction.

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

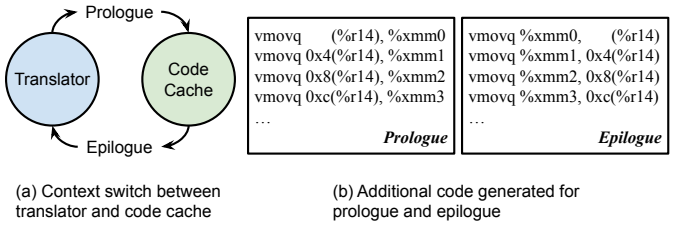

Fig. 5. Exploiting host SIMD registers to emulate guest general-purpose registers in cross-ISA virtualization.

TABLE II
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	RISC-V	PowerPC
QEMU [22]	G	N	N	G
Dyninst [28]	G	-	-	-
Dolphin [21]	G	G	-	G
Valgrind [29]	G	G	-	G
DynamoRIO [30]	G	G	-	-
AndroidEmu [19]	G	-	-	-

1) Host SIMD Registers used by the Translator: The first obstacle is that the host SIMD registers can also be used by the translator itself. Typically, the translator is written in high-level programming languages, such as C/C++, and compiled by a native compiler, e.g., GCC. Given the availability of SIMD extension on the host machine and the SIMD-related optimizations in the native compiler, it is foreseeable that some SIMD instructions may be generated during the compilation of the source code 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. Even worse, it is almost impossible to predict which SIMD registers will be used by the translator, as it is completely dependent on the source code of the translator and the compiler that is used to compile the source code.

To address this issue, we instead 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 prologue and epilogue, as depicted in Fig. 5(a). In essence, the prologue and epilogue are two pieces of host binary code that save and restore the contexts of the corresponding execution environments. In our design, we still keep the host memory locations that are used to emulate guest general-purpose registers, but use them only in the translator environment. At the same time, we further extend the prologue and epilogue to incorporate the usage of host SIMD registers in the code cache environment.

Fig. 5(b) shows an example of the additional code appended to the prologue and epilogue. In particular, we restore/save the host SIMD registers from/to the corresponding host memory locations in the prologue and epilogue, respectively. Note that there is no need to save/restore the host SIMD registers for the translator environment as these registers are typically *caller-saved* registers and, therefore, additional code will be generated automatically by the compiler of the translator to save them before entering into the prologue or restore them after exiting from the epilogue. Through this way, we can resolve the aforementioned conflict caused by the usage of host SIMD registers in the translator.

2) Host SIMD Registers used in the Translated Host Binary Code: Apart from the translator, the host binary code 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 we map guest general-purpose registers to the same host SIMD registers. Therefore, we need to tackle this issue.

To this end, we conduct a comprehensive study on a variety of open-source DBT systems. This study attempts to reveal 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 even though the guest ISA is typically the same as the host ISA. We manually examine the source code of each system to understand how it works and investigate the SIMD-related implementation details in the source code. Some DBT systems support multiple host ISAs, we then check all host ISAs with SIMD extensions. To validate the conclusions, we also develop some toy guest programs to test the systems. The details are omitted here due to space limitations.

TABLE II 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 and thus can be exploited by us to emulate guest general-purpose registers, especially when guest applications have very few SIMD instructions.

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);
```

As indicated by the names of the routines, they are expected to be invoked at the start and end points of the usage of

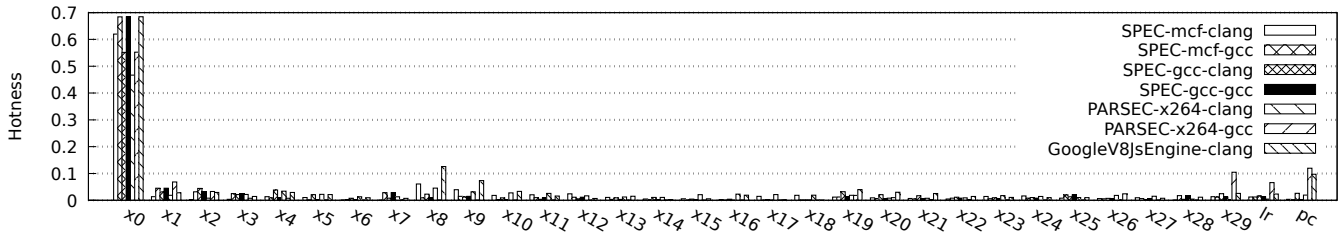

Fig. 6. Hotness of AArch64 registers across different applications and different compilers, e.g., "SPEC-mcf-clang" means the SPEC-CPU2017 *mcf* benchmark compiled by LLVM. Here the hotness of a register is defined as the percentage of occurrences of this register in dynamically-executed instructions.

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. In this way, we can remove the conflicts caused by the utilization of SIMD registers in the translated host binary code.

3) Which Guest Registers should be Mapped: The final challenge we face when exploiting SIMD registers for guest register emulation is that the host architecture does not have sufficient SIMD registers to emulate *all* guest general-purpose registers. This is possible if the host architecture is equipped with an old generation of the SIMD extension. For example, the AArch64 architecture has 32 general-purpose registers while the Intel x86 SSE extension has only 16 SIMD registers. So the problem here is which guest registers should be selected to be emulated using the limited host SIMD registers. A naïve solution is to select guest registers randomly. Though it is simple, this solution may only achieve sub-optimal performance improvement, given that different guest registers are very likely to be accessed in quite different frequencies.

To overcome this challenge, we propose to map guest registers based on the *hotness* of the guest registers. This is inspired by the observation that by mapping the most frequently accessed guest registers to host SIMD registers, we can remove the host load/store instructions caused by emulating these guest registers and thus achieve better performance improvement. A potential concern of this solution 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 some applications. Fig. 6 presents the results. As shown in the figure, the hotness trends of general-purpose registers are quite steady across different applications and even different compilers. This demonstrates the legitimacy and practicability of the proposed hotness-based selection scheme of guest registers.

With the above challenges addressed, we are able to map guest general-purpose registers to host SIMD registers and achieve better performance efficiency. In fact, this emulation mechanism also brings in many optimization opportunities for us to generate more optimized host binary code. We will discuss more details in the following section.

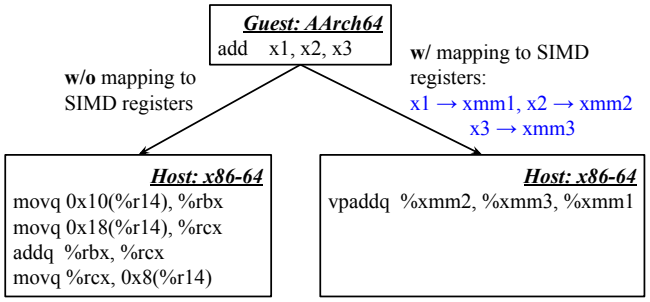

Fig. 7. Exploiting host SIMD operands to generate efficient host code.

B. 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 two common features that are available on most popular SIMD extensions. We next describe them in details, as well as how to exploit them in DBT systems.

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 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

Algorithm 1: Host instruction selection for code 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  foreach guest register r accessed by I do
3  |  if Query(T, I, r) ≠ host SIMD register then
4  |  |  flag ← false;
5  |  |  break;
6  |  end
7  end
8  if flag then
9  |  return SIMD;
10 else
11 |  return REGULAR;
12 end
```

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 instruction, 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 in the translator 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 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. 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.

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 perform *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 regular instructions, which are translated from multiple guest instructions accordingly, with a single SIMD instruction.

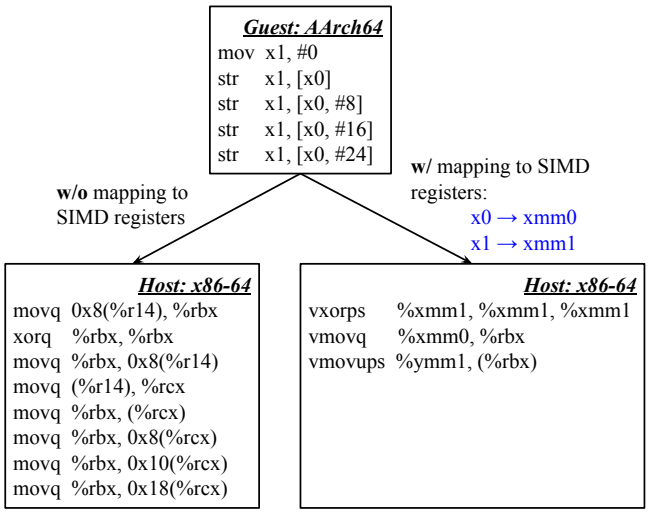

Fig. 8. Exploiting host SIMD opcodes to generate efficient host code.

Fig. 8 illustrates such an example. As shown in the figure, if the guest general-purpose registers are not emulated by host SIMD registers, eight host instructions are generated to emulate the semantics of 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.

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 here: (1) the identified guest instructions should perform the same operation; (2) the operation is supported by host SIMD instructions. To this end, each time 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. If yes, the corresponding guest instructions will be marked. The translator will then skip these guest instructions so that they can be translated to the host SIMD instruction. Our exploitation only covers common simple operations to avoid potential heavy translation overhead caused by sophisticated dynamic dependence analyses. In fact, complicated operations are often examined by compilers at the static compilation stage and exploited through auto-vectorization techniques.

IV. Implementation

We have implemented the proposed exploitation of SIMD extensions in a prototype based on QEMU [22] (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 takes AArch64 as the guest ISA and x86-64 as the host ISA. At the time of paper submission, we are working on the support for the RISC-V guest. We plan to commit our implementation into the QEMU project [31] in order to facilitate the following related research activities.

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 Ops and then the TCG Ops are translated to host binary code. In 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.

A. 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 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.

B. Condition 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. If 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.

C. Handling Helper Function Calls

The major task of TCG is to translate guest instructions into host instructions with equivalent semantics. However, in some cases, due to the complexity of guest instructions and the limitation of host ISAs, it requires significant engineering efforts or is rather difficult to generate semantically-equivalent host instructions. To address this issue, TCG introduces *helper functions* to emulate complicated and obscure guest instructions. These helper functions are typically developed in high-level programming languages for simplicity and expected to be invoked directly from code cache through host *call* instructions. It is worth noting 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. In particular, before a helper function is called, the values in host SIMD registers are saved to the corresponding host memory locations that are used to emulate guest registers. And after the helper function returns, the saved values are then restored to host SIMD registers. Although this solution has no correctness issue, it may introduce extremely high performance overhead, especially when the helper functions are invoked frequently and there are many host SIMD registers to save and restore.

Our investigations on the source code and binary code of helper functions reveal that most helper functions are self-contained and use very few host SIMD registers. More importantly, the source code of help functions is quite stable. This inspires 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.

V. Experimental Results

To evaluate the effectiveness of the proposed exploitation of SIMD extensions, we conduct comprehensive experiments on a wide range of benchmark applications, including PARSEC [32] (version 3.0), SPEC-CPU2017 [33], and Octane JavaScript applications [34] (verison 2.0). These applications cover various problem domains.

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. We compile the benchmark applications using the `pthread` configuration to test the bench-

TABLE III
CONFIGURATIONS OF OUR EVALUATION PLATFORM

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	

marks. Our evaluation includes all benchmark applications except *bodytrack*, *raytrace* and *vips* due to the compilation failures to AArch64 binaries.

SPEC-CPU2017 is an industry-standardized CPU-intensive benchmark suite. It has been widely used to measure and compare computer systems, in particular CPU processors. Our evaluation uses the *speed* configuration, which adopts the execution time as the performance metric. 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 [35]. 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 during the compilation process. TABLE III shows the detailed configurations of our evaluation platform, which is exclusively occupied during the evaluation. Also, 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.

A. PARSEC

Fig. 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 performance speedup. This demonstrates the capability of our SIMD exploitation to optimize the performance of cross-ISA virtualization.

We further count dynamic host instructions executed in QEMU with and without our SIMD exploitation. The results

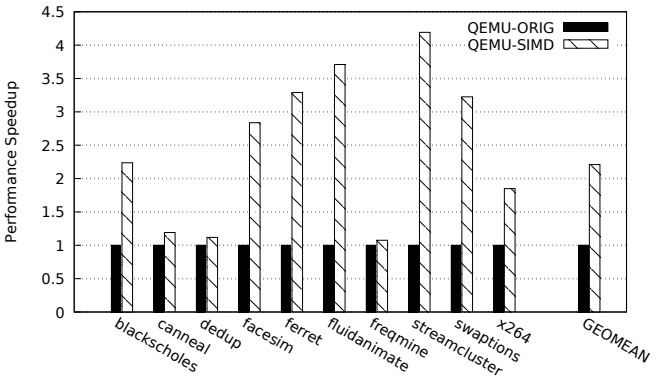

Fig. 9. Performance speedup achieved by our SIMD exploitation, denoted by QEMU-SIMD, for PARSEC benchmarks with 16 threads. The original QEMU, denoted by QEMU-ORIG,is the performance baseline.

TABLE IV
DYNAMIC HOST INSTRUCTIONS REDUCED BY OUR SIMD EXPLOITATION.

	QEMU-ORIG	QEMU-SIMD	Reduced
blackscholes	1.48×10^{13}	3.33×10^{13}	55.54%
canneal	2.32×10^{11}	3.32×10^{11}	30.00%
dedup	3.21×10^{11}	3.97×10^{11}	19.21%
facesim	1.68×10^{12}	5.15×10^{12}	67.37%
ferret	4.58×10^{11}	1.32×10^{12}	65.19%
fluidanimate	1.24×10^{14}	4.04×10^{13}	67.41%
freqmine	1.73×10^{13}	1.55×10^{13}	10.65%
streamcluster	1.43×10^{14}	4.07×10^{13}	71.46%
swaptions	7.96×10^{13}	2.94×10^{13}	63.00%
x264	5.75×10^{13}	3.56×10^{13}	38.10%

are presented in TABLE IV. As shown in the table, our SIMD exploitation requires less host instructions to emulate guest applications. This is because many host memory movement instructions are removed due to the mapping from guest general-purpose registers to host SIMD registers. From Fig. 9 and TABLE IV, we can also observe that the percentage of reduced host instructions is generally in proportion to the performance improvement.

Fig. 10 shows the scalability results for PARSEC benchmarks with different numbers of threads. The performance baseline here is the single-thread performance of the original QEMU without our SIMD exploitation. As shown in the figure, the performance of the original QEMU is improved when the number of threads increases. More importantly, we can clearly see from the figure that our SIMD exploitation enables QEMU to achieves better scalability results. A potential concern is that the performance of QEMU with our SIMD exploitation decreases for some benchmarks, e.g., *blackscholes*, when the number of threads increases from eight to sixteen. The reason is that the CPU processor of our evaluation platform has only eight physical cores and thus the contention on the SIMD hardware resources when there are more than eight threads may lead to harmful impact on the performance results. Overall, our exploitation is able to leverage more SIMD hardware resources to obtain better scalability efficiency.

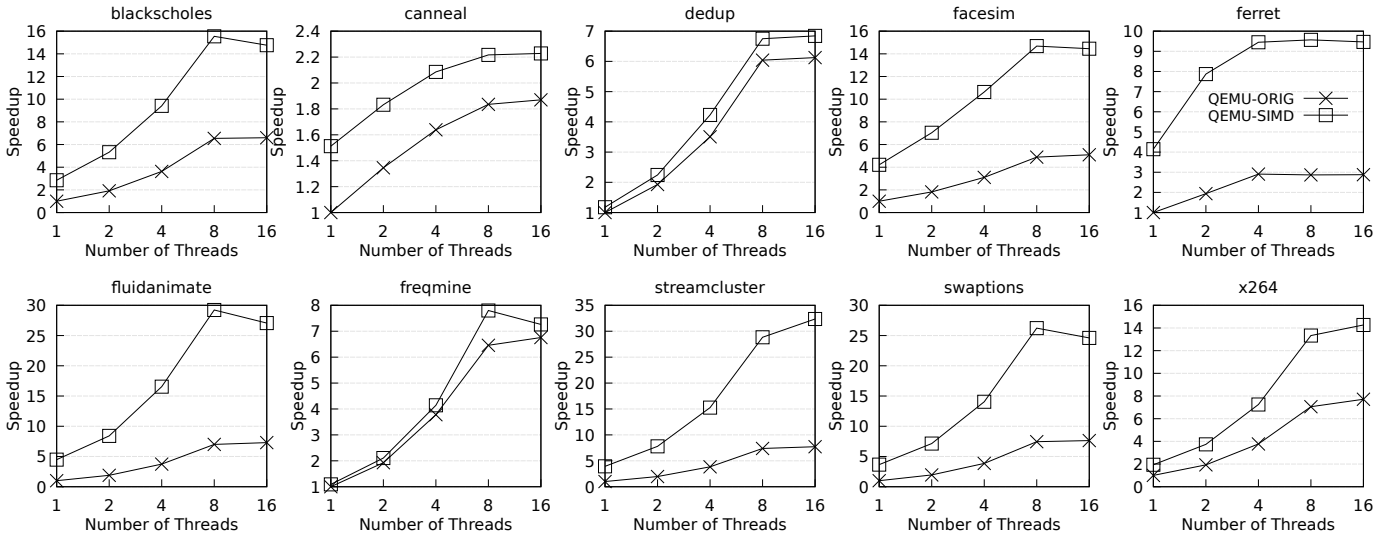

Fig. 10. Scalability improved by our SIMD exploitation for PARSEC benchmarks with the single-thread performance of original QEMU as the baseline.

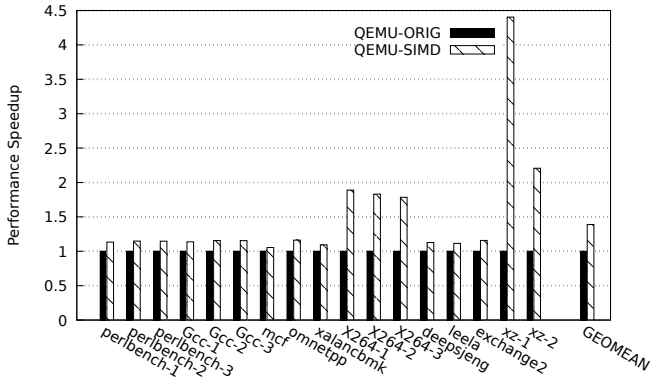

Fig. 11. Performance speedup achieved by our SIMD exploitation for SPEC-CPU2017 *intspeed* benchmarks. The baseline is the original QEMU. All benchmarks are evaluated with one thread except *xz*, which has 16 threads.

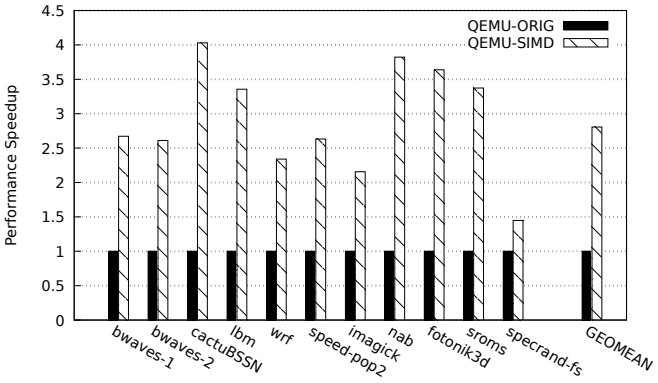

Fig. 12. Performance speedup achieved by our SIMD exploitation for SPEC-CPU2017 *fpspeed* benchmarks. The baseline is the original QEMU. All benchmarks are evaluated with 16 threads.

B. SPEC-CPU2017

Fig. 11 shows the performance speedup achieved by our SIMD exploitation for SPEC-CPU2017 intspeed benchmarks compiled to AArch64. Here, the performance baseline is the original QEMU without our 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 benchmark applications. This is 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, especially the last level cache 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. This is similar to the scalability improvements in PARSEC benchmarks. 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, Fig. 12 shows the performance speedup achieved by the proposed SIMD exploitation for SPEC-CPU2017 fpspeed benchmarks compiled to AArch64. As illustrated in the figure, with our SIMD exploitation, QEMU attains performance improvements for all evaluated fpseed 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.

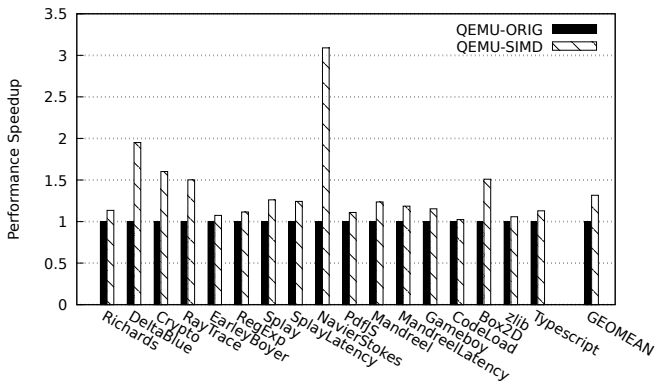

Fig. 13. Performance speedup achieved by our SIMD exploitation for Octane JavaScript applications running on the Google V8 JavaScript engine.

C. JavaScript Applications

Fig. 13 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.

VI. Related Work

A. SIMD Optimizations

A considerable amount of optimizations have been proposed to exploit SIMD extensions available on commercial CPU processors [1]–[5], [8]–[11]. These SIMD optimizations often target different application domains. For instance, it has been a long history to utilize SIMD extensions in database systems [3]–[5]. Zhou et al. leverage SIMD instructions to implement common database operations, including sequential scans, aggregation, index operations, and joins [3]. 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 [4]. Besides, SIMD extensions have been adopted to optimize graph and sorting algorithms [1], [2]. Moreover, language runtimes have integrated SIMD support for enhanced efficiency [8], [9].

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 [6], [7], [36]–[39]. In order to generate SIMD instructions, these mechanisms typically employ program analysis techniques and sophisticated vectorization algorithms to recognize data-level parallelism and resolve potential 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 [36].

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 any 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.

B. Optimizing DBT

Given the critical role of DBT in cross-ISA virtualization, many schemes have been proposed to optimize DBT. Some of these optimizations focus on SIMD extensions [40]–[43]. In particular, these optimizations improve the performance of DBT systems for guest applications that have SIMD instructions by generating more efficient host binary code. A noticeable example is saSLP [40], 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. 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 does not have such a requirement, 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 optimizations [44]–[47]. 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.

VII. Conclusion

SIMD extensions have become a permanent and pivotal hardware component in modern CPU processors. Existing optimizations exploit SIMD extensions by only exploring fine-grained data-level parallelism in target 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. Different from existing SIMD-related optimizations, our exploitation does not rely on data-level parallelism in guest applications. Instead, we take 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 broadly-used cross-ISA virtualization platform, QEMU. Experimental results on a wide range of benchmark applications demonstrate that the proposed exploitation is capable of achieving significant performance improvements. We believe this paper provides us a new perspective to rethink the current way to exploit hardware SIMD extensions.

REFERENCES

[1] S. Han, L. Zou, and J. X. Yu, "Speeding Up Set Intersections in Graph Algorithms Using SIMD Instructions," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1587–1602. [Online]. Available: https://doi.org/10.1145/3183713.3196924

[2] K. Hou, H. Wang, and W.-c. Feng, "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*, ser. ICS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 383–392. [Online]. Available: https://doi.org/10.1145/2751205.2751247

[3] J. Zhou and K. A. Ross, "Implementing Database Operations Using SIMD Instructions," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 145–156. [Online]. Available: https://doi.org/10.1145/564691.564709

[4] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking SIMD Vectorization for In-Memory Databases," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1493–1508. [Online]. Available: https://doi.org/10.1145/2723372.2747645

[5] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz, "Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask," *Proc. VLDB Endow.*, vol. 11, no. 13, p. 2209–2222, Sep. 2018. [Online]. Available: https://doi.org/10.14778/3275366.3284966

[6] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When Polyhedral Transformations Meet SIMD Code Generation," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 127–138. [Online]. Available: https://doi.org/10.1145/2491956.2462187

[7] S. S. Baghsorkhi, N. Vasudevan, and Y. Wu, "FlexVec: Auto-Vectorization for Irregular Loops," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 697–710. [Online]. Available: https://doi.org/10.1145/2908080.2908111

[8] A. Stojanov, I. Toskov, T. Rompf, and M. Püschel, "SIMD Intrinsics on Managed Language Runtimes," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 2–15. [Online]. Available: https://doi.org/10.1145/3168810

[9] C. Eidt and T. Gooding, "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*, ser. CGO 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 229–241. [Online]. Available: https://doi.org/10.1145/3368826.3377926

[10] E. Georganas, S. Avancha, K. Banerjee, D. Kalamkar, G. Henry, H. Pabst, and A. Heinecke, "Anatomy of High-Performance Deep Learning Convolutions on SIMD Architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[11] B. Akin, Z. A. Chishti, and A. R. Alameldeen, "ZCOMP: Reducing DNN Cross-Layer Memory Footprint Using Vector Extensions," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 126–138. [Online]. Available: https://doi.org/10.1145/3352460.3358305

[12] Intel Corporation, "Intel SPMD Program Compiler: An open-source compiler for high-performance SIMD programming on the CPU," 2019, https://ispc.github.io.

[13] LLVM, "Auto-Vectorization in LLVM," 2020, https://llvm.org/docs/Vectorizers.html.

[14] OpenMP, "OpenMP Application Program Interface Version 4.0," 2013, https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.

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

[16] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "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*, ser. CGO '03. USA: IEEE Computer Society, 2003, p. 15–24.

[17] M. DeVuyst, A. Venkat, and D. M. Tullsen, "Execution migration in a heterogeneous-isa chip multiprocessor," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: Association for Computing Machinery, 2012, p. 261–272. [Online]. Available: https://doi.org/10.1145/2150976.2151004

[18] W. Wang, P.-C. Yew, A. Zhai, S. McCamant, Y. Wu, and J. Bobba, "Enabling Cross-ISA Offloading for COTS Binaries," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 319–331. [Online]. Available: https://doi.org/10.1145/3081333.3081337

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

[20] Q. Yang, Z. Li, Y. Liu, H. Long, Y. Huang, J. He, T. Xu, and E. Zhai, "Mobile gaming on personal computers with direct android emulation," in *The 25th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3300061.3300122

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

[22] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. USENIX ATC '05. USA: USENIX Association, 2005, p. 41.

[23] A. Yazdanbakhsh, K. Samadi, N. S. Kim, and H. Esmaeilzadeh, "GANAX: A Unified MIMD-SIMD Acceleration for Generative Adversarial Networks," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 650–661. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00060

[24] J. P. Patwardhan, V. Johri, C. Dwyer, and A. R. Lebeck, "A Defect Tolerant Self-Organizing Nanoscale SIMD Architecture," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: Association for Computing Machinery, 2006, p. 241–251. [Online]. Available: https://doi.org/10.1145/1168857.1168888

[25] A. Hagiescu and W.-F. Wong, "Co-Synthesis of FPGA-Based Application-Specific Floating Point Simd Accelerators," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 247–256. [Online]. Available: https://doi.org/10.1145/1950413.1950459

[26] V. J. Reddi, D. Connors, R. Cohn, and M. D. Smith, "Persistent Code Caching: Exploiting Code Reuse Across Executions and Applications," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '07. USA: IEEE Computer Society, 2007, p. 74–88. [Online]. Available: https://doi.org/10.1109/CGO.2007.29

[27] W. Wang, P.-C. Yew, A. Zhai, and S. McCamant, "A General Persistent Code Caching Framework for Dynamic Binary Translation (DBT)," in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '16. USA: USENIX Association, 2016, p. 591–603.

[28] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn Parallel Performance Measurement Tool," *Computer*, vol. 28, no. 11, p. 37–46, Nov. 1995. [Online]. Available: https://doi.org/10.1109/2.471178

[29] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 89–100. [Online]. Available: https://doi.org/10.1145/1250734.1250746

[30] D. L. Bruening and S. Amarasinghe, "Efficient, Transparent, and

Comprehensive Runtime Code Manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, USA, 2004.
[31] QEMU, "the FAST! processor emulator," 2020, https://qemu.org.
[32] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 72–81. [Online]. Available: https://doi.org/10.1145/1454115.1454128
[33] Standard Performance Evaluation Corporation, "SPEC CPU 2017," 2020, https://www.spec.org/cpu2017.
[34] Google, "The JavaScript Benchmark Suite for the modern web," 2020, https://developers.google.com/octane.
[35] ——, "V8 JavaScript engine," 2020, https://v8.dev.
[36] V. Porpodas, "SuperGraph-SLP Auto-Vectorization," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 330–342.
[37] V. Porpodas, R. C. O. Rocha, E. Brevnov, L. F. W. Góes, and T. Mattson, "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*, ser. CGO 2019. IEEE Press, 2019, p. 206–216.
[38] J. Huh and J. Tuck, "Improving the Effectiveness of Searching for Isomorphic Chains in Superword Level Parallelism," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 718–729. [Online]. Available: https://doi.org/10.1145/3123939.3124554
[39] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks, "Vapor SIMD: Auto-Vectorize Once, Run Everywhere," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '11. USA: IEEE Computer Society, 2011, p. 151–160.
[40] Y. Liu, D. Hong, J. Wu, S. Fu, and W. Hsu, "Exploiting Asymmetric SIMD Register Configurations in ARM-to-x86 Dynamic Binary Translation," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 343–355.
[41] Y. Pang, R. Lyerly, and B. Ravindran, "Cross-ISA Execution of SIMD Regions for Improved Performance," in *Proceedings of the 12th ACM International Conference on Systems and Storage*, ser. SYSTOR '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 55–67. [Online]. Available: https://doi.org/10.1145/3319647.3325832
[42] S.-Y. Fu, D.-Y. Hong, Y.-P. Liu, J.-J. Wu, and W.-C. Hsu, "Dynamic Translation of Structured Loads/Stores and Register Mapping for Architectures with SIMD Extensions," in *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 31–40. [Online]. Available: https://doi.org/10.1145/3078633.3081029
[43] C. Mendis, A. Jain, P. Jain, and S. Amarasinghe, "Revec: Program Rejuvenation through Revectorization," in *Proceedings of the 28th International Conference on Compiler Construction*, ser. CC 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 29–41. [Online]. Available: https://doi.org/10.1145/3302516.3307357
[44] A. D'Antras, C. Gorgovan, J. Garside, and M. Luján, "Low Overhead Dynamic Binary Translation on ARM," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 333–346. [Online]. Available: https://doi.org/10.1145/3062341.3062371
[45] W. Wang, S. McCamant, A. Zhai, and P.-C. Yew, "Enhancing Cross-ISA DBT Through Automatically Learned Translation Rules," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 84–97. [Online]. Available: https://doi.org/10.1145/3173162.3177160
[46] C. Song, W. Wang, P.-C. Yew, A. Zhai, and W. Zhang, "Unleashing the Power of Learning: An Enhanced Learning-Based Approach for Dynamic Binary Translation," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USA: USENIX Association, 2019, p. 77–89.
[47] T. Spink, H. Wagstaff, and B. Franke, "A Retargetable System-Level DBT Hypervisor," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USA: USENIX Association, 2019, p. 505–520.