Exploiting SIMD Asymmetry in ARM-to-x86 Dynamic Binary Translation

YU-PING LIU, National Taiwan University, Taiwan

DING-YONG HONG and JAN-JAN WU, Academia Sinica, Taiwan

SHENG-YU FU and WEI-CHUNG HSU, National Taiwan University, Taiwan

2

Single instruction multiple data (SIMD) has been adopted for decades because of its superior performance and power efficiency. The SIMD capability (i.e., width, number of registers, and advanced instructions) has diverged rapidly on different SIMD instruction-set architectures (ISAs). Therefore, migrating existing applications to another host ISA that has fewer but longer SIMD registers and more advanced instructions raises the issues of asymmetric SIMD capability. To date, this issue has been overlooked and the host SIMD capability is underutilized, resulting in suboptimal performance. In this article, we present a novel binary translation technique called spill-aware superword level parallelism (saSLP), which combines short ARMv8 instructions and registers in the guest binaries to exploit the x86 AVX2 host's parallelism, register capacity, and gather instructions. Our experiment results show that saSLP improves the performance by $1.6\times$ ($2.3\times$) across a number of benchmarks and reduces spilling by 97% (99%) for ARMv8 to x86 AVX2 (AVX-512) translation. Furthermore, with AVX2 (AVX-512) gather instructions, saSLP speeds up several data-irregular applications that cannot be vectorized on ARMv8 NEON by up to $3.9\times$ ($4.2\times$).

CCS Concepts: • **General and reference** → **Design; Performance**; • **Computer systems organization** → **Single instruction, multiple data**; • **Software and its engineering** → **Dynamic compilers**;

Additional Key Words and Phrases: Dynamic binary translation, SIMD, SLP vectorization

ACM Reference format:

Yu-Ping Liu, Ding-Yong Hong, Jan-Jan Wu, Sheng-Yu Fu, and Wei-Chung Hsu. 2019. Exploiting SIMD Asymmetry in ARM-to-x86 Dynamic Binary Translation. *ACM Trans. Archit. Code Optim.* 16, 1, Article 2 (February 2019), 24 pages.

<https://doi.org/10.1145/3301488>

This article is an extension of a paper presented at PACT'17 (Liu et al. 2017). The additional material includes a new algorithm to vectorize data-irregular loops in DBTs using gather instructions, a new approach to vectorize loop reductions in DBTs, details about the implementation, and extensive experiment results.

This work is supported in part by the Ministry of Science and Technology of Taiwan under grant number MOST-106-2218-E-002-040. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Ministry of Science and Technology of Taiwan.

Authors' addresses: Y.-P. Liu, National Taiwan University, No. 1, Sec. 4, Roosevelt Rd., Daan Dist. Taipei, 10617, Taiwan; email: r04922005@csie.ntu.edu.tw; D.-Y. Hong and J.-J. Wu, Academia Sinica, No. 128, Sec. 2, Academia Rd., Nankang Dist. Taipei, 11529, Taiwan; emails: {dyhong, wuj}@iis.sinica.edu.tw; S.-Y. Fu and W.-C. Hsu, National Taiwan University, No. 1, Sec. 4, Roosevelt Rd., Daan Dist. Taipei, 10617, Taiwan; emails: {d03922013, hsuwc}@csie.ntu.edu.tw.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Association for Computing Machinery.

1544-3566/2019/02-ART2

<https://doi.org/10.1145/3301488>

1 INTRODUCTION

Dynamic binary translation (DBT) is a widely used approach that realizes cross-instruction-set architecture (ISA) migration by emulating the application binaries of one guest ISA on a host machine with a different ISA. The advantage of not requiring source recompilation makes the use of DBTs critical in migrating legacy or proprietary applications across different ISAs. For example, many Android applications include ARM native code for performance-critical parts. With the DBT technique, the widely used Android emulators can run such ARM binaries on x86 platforms for application streaming, development, and security analysis. As the efficiency of the emulation process is crucial to the success of cross-ISA DBTs, optimizing the translated code is a key issue in the design of cross-ISA DBTs.

Single instruction multiple data (SIMD) provide both execution and power efficiency by exploiting fine-grained data parallelism in a wide range of applications. As a result, all major ISAs have adopted SIMD as extensions; for example, NEON for ARM, SSE/AVX for x86, and AltiVec for PowerPC. Moreover, SIMD architectures have been improved through the introduction of extra wider registers and advanced instructions. For instance, the Intel x86 architecture has evolved from SSE, which has 16 16B registers with elementary vector operations, to the current AVX-512 where 32 64B registers can be used with more advanced instructions such as gather/scatter (Intel Corp. 2018b). Meanwhile, the number of 16B registers in NEON has increased from 16 in ARMv7 to 32 in ARMv8 without extending the register width and supporting gather/scatter (ARM Ltd. 2015). The evolution has enabled applications to explore more parallelism. However, it has also led to divergent widths, number of registers, and advanced instructions on different ISAs.

The growing importance of SIMD has reinforced the need to translate SIMD binaries efficiently. A number of methods (Fu et al. 2015; Li et al. 2006; Michel et al. 2011) have been proposed to establish feasible mappings of guest SIMD registers and instructions to the host ones. They map a guest SIMD register directly to a host register with the *same* width, and map a guest SIMD instruction to equivalent host instructions that operate on the *same* number of elements. More recent approaches (Hallou et al. 2016, 2015; Hong et al. 2016) use loop vectorization to exploit more parallelism but still map *one* guest SIMD register to *one* host SIMD register. However, existing works fail to address the challenges raised by asymmetric SIMD capability, in which the host may have fewer but longer SIMD registers than the guest. These methods either (1) underutilize the host's parallelism or (2) they lead to register spilling because of the excessive number of host registers required for mapping so that only suboptimal performance is realized.

To solve the problem, we seek to exploit longer host SIMD registers for both parallelism and register capacity. To this end, we propose a new cross-ISA DBT technique called spill-aware superword level parallelism (saSLP). Given a guest binary loop that has been optimized according to the width and number of guest SIMD registers, saSLP combines multiple guest SIMD instructions and registers to form a longer one on the host. In addition, saSLP exploits the host's advanced SIMD instructions to vectorize loops that cannot be vectorized on the guest. These increase the number of elements that can be processed and stored in each host SIMD register and thereby improves the overall performance.

The design of saSLP faces several challenges. First, combining guest instructions involves a vectorization legality issue because we must ensure that there is no data dependency for reordering memory operations. This issue is more challenging for DBTs than for static compilers since DBTs work on a smaller scope (e.g., traces) and a great deal of source-level information is lost in binaries. Hence, with limited information, DBTs may not always be able to determine data dependency at the time of translation. Second, though combining guest registers can reduce the number of host registers used for mapping, access to an individual guest register requires unpacking the

host register, which results in additional costs. Thus, arranging a suitable trade-off between such overheads and the benefits of saving host registers is a critical issue.

The key contributions of this article are as follows:

- We identify the issues of asymmetric SIMD capability and propose saSLP as an effective solution. It combines multiple guest instructions to form a longer SIMD instruction that can be used to exploit the host's parallelism.
- We propose a novel spill-aware cost model that can determine whether combining guest registers would be profitable by weighing the extra unpacking overheads and the benefits of conserving host registers.
- We present the vectorization of data-irregular loops by exploiting SIMD gather instructions and propose a new algorithm to vectorize loop reductions in DBTs, where the typical methods used in static compilers do not work.
- We evaluated saSLP with ARMv8 to x86 AVX2 (and AVX-512) translations. Benchmark results demonstrate that saSLP improves the performance by 1.6 $\times$ (resp., 2.3 $\times$) and reduces 97% (resp., 99%) of register spilling, on average. By utilizing AVX2 (AVX-512) gather instructions, saSLP achieves 1.9 $\times$ (resp., 1.74 $\times$) average speedup on a set of data-irregular applications.

The remainder of this article is organized as follows. Section 2 provides an overview of cross-ISA DBTs and explains how asymmetric SIMD capacity causes problems. We describe our approach, saSLP, in Section 3 and then report the evaluation results in Section 4. Section 5 deals with related work and Section 6 contains our concluding remarks.

2 BACKGROUND AND MOTIVATION

One of the core techniques in cross-ISA DBTs is mapping guest registers and instructions to effective host ones. We start with an overview of such mappings and then explain how the mappings underutilize the host's SIMD capability.

2.1 Overview of Cross-ISA Dynamic Binary Translation

Most cross-ISA DBTs translate guest binaries in the granularity of a translation code fragment and maintain a guest architectural state in the host memory for emulation. To reduce memory access, typical retargetable and cross-ISA DBTs (Baraz et al. 2003; Bellard 2005; Dolan-Gavitt et al. 2015; Fu et al. 2015) map each guest SIMD register in the architectural state to a virtual register with the *same* width within a translation code fragment. Thus, they have to synchronize modified virtual registers back to the guest state at the exit points. Similarly, guest SIMD instructions are mapped to equivalent machine-independent instruction register (IR) that operates on the *same* number of vector elements. In the final step, the DBT back end allocates host registers to mapped virtual registers and converts mapped IRs to the host binary. Figure 1 illustrates the translation of an ARMv8 NEON vector addition of two 8B integers.

2.2 An Example of Asymmetric SIMD Capability

As the width and number of SIMD registers have become more divergent between ISAs, compilers need to consider ISA-dependent SIMD registers when performing optimization. For example, many modern compilers, such as GCC, LLVM, and ICC, vectorize and unroll loops according to the SIMD registers of target ISAs to exploit both data and instruction level parallelism (ILP). The example in Figure 2 illustrates the procedure. For ease of presentation, we scale down the number of SIMD registers in ARMv8 NEON and x86 AVX2 from 32 and 16 to 6 and 3, respectively.

Fig. 1. An example of register and instruction mapping in DBT.

Fig. 2. Limitations of typical mapping methods and the saSLP solution for SIMD asymmetry.

Next, we explain how compilers optimize the loop in Figure 2(a) for NEON. First, the compiler vectorizes the loop by maximizing the number of elements processed in a NEON register. For example, in Figure 2(b), each of the 16B registers ($v0-v4$) holds four 4B floating-point numbers. Second, to achieve better ILP, the compiler unrolls and interleaves the loop according to the six available NEON registers. The loop is unrolled only once because a larger unrolling factor would require more than six registers and cause spilling. The final compiled binary is shown in Figure 2(b). Instructions from the first and second unrolled iteration are marked ①-⑤ and ①-⑤, respectively.

For cross-ISA DBTs, such ISA-dependent optimizations cause problems in the typical register and instruction mapping discussed earlier. This is because a binary loop optimized for the guest could become non-optimal on the host, which has a different width and number of SIMD registers. For instance, to translate the NEON binary in Figure 2(b), typical methods map each 16B NEON register ($v0-v4$) to one virtual register ($\%v0-\%v4$) with the same width, resulting in a total of five live ranges in Figure 2(c). All NEON instructions are mapped to IR instructions that also operate on four floating-point numbers. As a result, the parallelism that can operate on eight floating-point numbers on the AVX2 host is underutilized. Furthermore, since the AVX2 host in this case has only three registers, the five overlapping live ranges in Figure 2(c) will cause register spilling in

the resulting AVX2 binary, and only half of the 32B AVX2 register capacity can be used to store a 16B NEON register.

To solve these problems, our saSLP algorithm transforms the IR in Figure 2(c) to that in Figure 2(d). First, it combines each of the five instructions ①-⑤ in the first loop iteration with ①-⑤ in the second unrolled iteration in a pairwise manner. Therefore, the combined instructions (e.g., ③ ③) in Figure 2(d) can now operate on eight floating-point numbers and thus fully utilize the AVX2 parallelism. Second, saSLP combines four 16B virtual registers (%v0-%v3) in Figure 2(c) to form two 32B virtual registers (%v02 and %v13) in Figure 2(d). As a result, register spilling will no longer occur because the number of overlapping live ranges is reduced to three by fully utilizing the AVX2 register capacity.

However, saSLP faces several challenges when performing this transformation. First, to combine the instructions in Figure 2(c), saSLP must ensure there is no data dependency between the reordered memory instructions. Yet, with the limited information in binaries, saSLP may not always be able to detect the dependency at the translation time. For example, the order of load ① and store ⑤ in Figure 2(c) is swapped in Figure 2(d), and they access the memory locations %x1 + 16 and %x2, respectively. Since the pointer values %x1 and %x2 are unknown at the time of translation, a conditional translation approach with a runtime check is required to avoid the execution of transformed code in Figure 2(d) when %x1 and %x2 are aliased (i.e., ① and ⑤ are dependent).

Second, although combining short registers in Figure 2(c) can save host registers, it is necessary to unpack the short register from the corresponding long register to access an individual combined short register. For example, the two modified long registers (%v02 and %v13) in Figure 2(d) need to be unpacked to four short registers (%v0-%v3) and then stored back in the guest state for synchronization. The overheads of extra unpacking might nullify the benefits of saving host registers. Therefore, a cost model that can precisely estimate the gain of saving host registers and the cost of extra data reordering is required to determine if short registers should be combined.

3 SPILL-AWARE SLP

In this section, we present our approaches for the above-mentioned transformation and then provide details of the solutions to the two challenges in Section 3.3 and Section 3.4, respectively. saSLP is an optimization algorithm designed for cross-ISA DBTs. First, we introduce the algorithm and then describe how the instructions are combined and how the registers are combined.

3.1 Overview

Figure 3 details the steps of the saSLP algorithm. The white boxes are related to combining guest instructions to form longer ones on the host; the highlighted parts are designed to address the two challenges mentioned earlier.

As shown in Figure 3, saSLP targets innermost loops captured by the DBTs, because they are usually the hottest (step 1). Then, saSLP collects short (i.e., scalar or vector) instructions in the loop body as seeds. Starting with those seeds, saSLP constructs a combining graph by traversing the use-define chain and then groups visited short instructions. The traversal stops at short instructions that cannot be combined. The grouped short instructions in each group are candidates for being combined with each other to form a long SIMD instruction on the host (step 2).

For each constructed combining graph, saSLP estimates the performance cost of the short and long forms (i.e., c_S , c_L in step 3). The long cost also includes the extra overheads of data movement (e.g., packing and unpacking). Then, saSLP credits the additional gain of saving host registers (i.e., c_R in step 3). If combining short instructions is profitable (step 4), saSLP versions the loop into an original short version and an optimized long version with runtime check when needed (steps 5 and 6). Finally, saSLP combines the grouped instructions to form longer SIMD instructions in the

Fig. 3. Overview of the saSLP algorithm.

ALGORITHM 1: Build Combining Graph**input:** A slice of seed instructions S .**output:** Combining graph G

```

1 Function construct_graph( $S$ )
2   Initialize  $G$  as an empty graph.
3   traverse_and_construct( $S$ )
4   return  $G$ 
5 Function traverse_and_construct( $S_1$ )
6   if  $S_1$  can be combined to form long SIMD then
7      $G.add\_group(S_1)$ 
8     foreach  $S_2 \in$  operand lists of  $S_1$  do
9       if  $S_2$  is a list of short instructions then
10        traverse_and_construct( $S_2$ )
11         $G.add\_edges(S_1, S_2)$ 
12      else
13         $G.add\_node\_and\_pack(S_1)$ 

```

```

① %v0 = load <4 x f32>* %x1
② %v1 = load <4 x f32>* %x2
③ %v3 = load <4 x f32>* [%x2 + 16]
④ %v0 = fmul <4 x f32> %v0, %v4
⑤ %v1 = fadd <4 x f32> %v1, %v0
⑥ store %v1, <4 x f32>* %x2
⑦ %v2 = load <4 x f32>* [%x1 + 16]
...
⑧ %v2 = fmul <4 x f32> %v2, %v4
⑨ %v3 = fadd <4 x f32> %v3, %v2
⑩ store %v3, <4 x f32>* [%x2 + 16]
...

```

(a) Input IR of the saSLP algorithm

(b) Data dependency graph

(c) Combining graph

Fig. 4. An example of saSLP combining graph.

long-version loop, and inserts additional data reordering instructions for packing and unpacking (step 7).

We use the simplified case of ARMv8 NEON to x86 AVX2 translation in Figure 2 as a running example. We use the terms *ShortVS* and *LongVS* to denote, respectively, the width of registers in the input IR (guest) and the output IR (host). In addition, *CF* (i.e., combining factor) denotes how many short instructions will be combined to form a long SIMD instruction. *CF* is computed as $\lfloor \text{LongVS} / \text{ShortVS} \rfloor$.

3.2 Combining Short Instructions to Form Long SIMD

The input of saSLP is the typically mapped IR of a guest binary loop, as described in Section 2.2. To construct the combining graphs, saSLP first collects consecutive short (i.e., scalar or vector) stores in the loop body to form seed chains. Starting from the seeds is a popular heuristic approach that modern compilers adopt (Lattner and Adve 2004). In the motivating example, where the *ShortVS* and *LongVS* is 16B and 32B, respectively, saSLP collects the two consecutive vector stores (⑤ and ⑩) in Figure 4(a) to form a seed chain of length two. Next, saSLP uses Algorithm 1 to construct the combining graphs in a bottom-up manner from the seeds with recursive traversal (lines 3 and 10) along the use-define chains (i.e., edges in Figure 4(b)). For a list of short instructions S_1 currently visited, the algorithm checks whether the short instructions in S_1 can be combined to form a long SIMD instruction (line 6).

Fig. 5. Overview of runtime check for possible dependency.

Short instructions in S_1 that can be combined (1) must have the same opcode and data type, (2) do not have a duplicated instruction in S_1 , and (3) can be scheduled in parallel. The last requirement ensures that all data flow and memory dependency are preserved after combining the short instructions. The reason is that the combined instructions are moved to the position of the resulting long SIMD instruction. Thus, to ensure that the combining is valid, saSLP marks reordered memory instructions that can be checked at runtime with techniques introduced in Section 3.3 as combinable. Others that cannot be checked at runtime are treated as dependent and non-combinable.

If short instructions in S_1 can be combined, they are grouped and inserted into the combining graph G (line 7 and solid-line rectangles in Figure 4(c)). Then, the traversal continues to trace along the use-define chains from S_1 to S_2 (i.e., following the arrows in Figure 4(b) in reverse order), where S_2 is a list of operands of S_1 (lines 8 and 9). Finally, edges connecting the newly inserted nodes of S_1 and S_2 are added to the combining graph (line 11 and arrows in Figure 4(c)).

For short instructions in S_1 that cannot be combined, the algorithm inserts a short node for each short instruction in S_1 into the combining graph and adds an extra packing operation node to pack the short nodes into a long SIMD node (line 13 and the dashed-line rectangle in Figure 4(c)). Therefore, the resulting long SIMD node can be used by other long SIMD nodes because they now have the same width. The traversal returns to the previous level at non-combinable S_1 .

In the motivating example, the graph construction starts the traversal from seeds $\langle 5, 5 \rangle$. As combining $\langle 5, 5 \rangle$ necessitates reordering of memory instructions $\langle 5, 5 \rangle$ and $\langle 1, 1 \rangle$, the algorithm records their pointers that can be checked at runtime and marks $\langle 5, 5 \rangle$ as combinable. The traversal follows the use-define chains and groups the add $\langle 4, 4 \rangle$, multiply $\langle 3, 3 \rangle$, and load instructions $\langle 1, 1 \rangle$, $\langle 2, 2 \rangle$. On the other hand, $\langle v4, v4 \rangle$ (i.e., the right-hand side operands of $\langle 3, 3 \rangle$) cannot be combined owing to the violation of requirement (2) and thus are packed into a long SIMD node. Figure 4(c) shows the final constructed combining graph.

3.3 Runtime Memory Aliasing Check

To detect memory dependency that cannot be determined at the translation time in DBTs, we propose a conditional translation with runtime check of pointer aliasing.

When saSLP encounters possible memory dependency, it versions the loop into an original short version (ShortLoop) and an optimized long SIMD version (LongLoop), and inserts checking code into the RTCheck block. This operation predicates the two versioned loops based on whether aliased pointers are detected at runtime (Figure 5(a)). Given a pair of may-alias pointers $\langle p, q \rangle$, saSLP first checks whether (1) the two pointers start from loop-invariant addresses (*StartAddr*) and move with constant strides (*StepSize*) in each loop iteration and (2) whether the input loop has a loop-invariant trip count (*TripCnt*). These ensure that saSLP can derive an invariant region from

$StartAddr$ to $StartAddr + TripCnt \times StepSize$ accessed by a pointer during the entire loop. Then, whether or not p and q are aliased can be verified at runtime by checking whether their regions overlap. Any overlapped region will force the execution to jump to the ShortLoop for legality. If $\langle p, q \rangle$ does not meet requirements (1) and (2), then they are treated as aliased for correctness.

We now consider the two memory instructions ⑤ and ①, which store and load 16B vectors with pointers $\%x2$ and $\%x1 + 16$, respectively, as shown in Figure 5(b). To detect the possible dependency (dashed-line arrow in Figure 5(c)), saSLP inserts checking code into RTCheck since both pointers move with 32B constant stride and the loop has an invariant trip count determined by $\%x0$. The code evaluates the accessed regions of $\%x2$ and $\%x1 + 16$ and determines whether the two regions overlap at runtime. The runtime check is further optimized by grouping pointers with constant distances and the same stride (e.g., the pointers of ⑤ and ⑥ share a 32B stride and differ by 16B) and performing the overlap tests between pointer groups instead of between individual pointers. Since saSLP targets unrolled guest loops where many pointers can be grouped, this optimization reduces the number of overlap tests significantly.

3.4 Spill-Aware Cost Model

To decide whether combining the grouped short instructions in the combining graph to form long SIMD instructions would be profitable, we devised a new spill-aware cost model to evaluate the overall performance cost of the combining graph.

To model the performance cost, the spill-aware cost model first categorizes cost factors into three classes: (1) the short cost c_S , which denotes the cost of executing original instructions with *ShortVS* width; (2) the long cost c_L , which denotes the cost of executing combined SIMD instructions with *LongVS* width as well as the extra data reordering overheads; and (3) the register saving gain c_R , which is the benefit of eliminating register spilling by combining multiple short registers. For each node N_i in the combining graph G , the cost model computes $c_S(N_i)$, $c_L(N_i)$, and $c_R(N_i)$, and then determines whether combining the grouped short instructions in G will be profitable if

$$\sum_{N_i \in G} [c_L(N_i) - c_R(N_i)] < \sum_{N_i \in G} c_S(N_i). \quad (1)$$

Our cost model largely reuses the existing instruction cost table in the DBT back-end compiler to compute c_S and c_L . Therefore, we focus on the computation of c_R . The core idea of c_R is assigning higher c_R to short instructions with live ranges that are likely to lead to register spilling. Thus, such live ranges have a greater chance of being combined and stored in long SIMD registers to avoid spilling. First, we define the following terms for an instruction n_i . (1) $Reg(n_i)$ denotes the physical register capacity occupied by n_i ; (2) $Int(n_i)$ denotes the maximal physical register capacity occupied by the live ranges that interfere with the live range of n_i ; (3) $Avail(n_i)$ denotes the total physical register capacity that can be used to store n_i . Hence, if $Int(n_i) \geq Avail(n_i)$, then n_i will lead to spilling because its live range interferes with too many live ranges and thus overwhelms the available register capacity on the host.

Given a list of CF short instructions $S_i = \langle s_1, \dots, s_{CF} \rangle$ that can be combined to form a long SIMD instruction l_i , the cost model computes the register saving gain $c_R(S_i)$ as follows:

$$\frac{1}{ShortVS} \sum_{s_j \in S_i} \begin{cases} Reg(s_j) - \frac{Reg(l_i)}{CF} & \text{if } Int(s_j) \geq Avail(s_j) \\ 0 & \text{, otherwise} \end{cases}. \quad (2)$$

In Equation (2), each short instruction s_j respectively occupies $Reg(s_j)$ and $Reg(l_i) / CF$ bytes (amortized over S_i) before and after the combining operation. Therefore, combining s_j can save $Reg(s_j) - Reg(l_i) / CF$ capacity. If s_j could lead to spilling (i.e., $Int(s_j) \geq Avail(s_j)$), the cost model rewards s_j with the saving in the unit of *ShortVS*.

Fig. 6. An example of saSLP cost model.

Next, we consider the motivating example in which the AVX2 host has three 32B YMM registers. For instance, the grouped short instructions ③ and ④ in Figure 6(a) have two 16B live ranges before being combined. Since the upper half of a 32B YMM register cannot be used when the lower half is used by ③ and ④, both ③ and ④ occupy a full YMM register (i.e., $Reg(③) = Reg(④) = 32$). After combining, the resulting long SIMD instruction ③④ occupies only a YMM register (i.e., $Reg(③④) = 32$). Also, both ③ and ④ interfere with a maximum of four live ranges, exceeding the available registers on the AVX2 host. Hence, according to Equation (2), combining them could save 32B of capacity, which is equal to two 16B *ShortVS* and implies that $c_R(\langle ③, ④ \rangle) = 2$.

To compute c_S and c_L , the cost model queries our DBT back-end compiler and finds that the cost of SIMD multiply instructions is 2 regardless of whether it operates on four or eight 4B floating-point numbers on the AVX2 host. Therefore, as shown in Figure 6(c), the cost model concludes that $c_S(\langle ③, ④ \rangle) = 2 + 2$ and $c_L(\langle ③, ④ \rangle) = 2$. Moreover, recall that all modified registers need to be synchronized back to the guest state. Thus, two extra unpacking instructions are required to extract $\%v0$ and $\%v2$ from the long SIMD instruction ③④, as shown in Figure 6(b). The additional cost of extraction is 2 based on the back-end interface. Therefore, unpacking ③④ adds an extra cost of 4 to the unpack row in Figure 6(c). The unpacking of ④④ also incurs an extra cost of 4. The cost model repeats the above computation on each node in the combining graph and produces the final cost table shown in Figure 6(c). As the performance cost of the long SIMD form ($c_L - c_R = 13$) is lower than the short one ($c_S = 18$), combining the grouped short instructions is profitable.

3.5 Combining Loop Reductions

In addition to the consecutive stores mentioned in Section 3.2, saSLP also selects a sequence of short (i.e., scalar or vector) reductions as seeds and combines them to form a long SIMD reduction. Combining reductions to exploit data parallelism is a critical issue because reductions are prevalent in applications (Hong et al. 2011; Huo et al. 2011). For example, PageRank, one of the most important graph applications, contains reductions in its hottest loop. As shown in Figure 7(a), for each node i in the given graph, the innermost loop iterates over the neighboring nodes of node i (i.e., nodes k) and accumulates their weighted rank values. The accumulated value is a typical reduction.

As mentioned in Section 2.2, compilers optimize loops according to target ISAs. To optimize the loop for the simplified ARMv8 guest that has six NEON registers, the compiler unrolls the loop once and produces the guest binary in Figure 7(b). The guest binary is then mapped to the IR in Figure 7(c), which is the input of saSLP. We use the PageRank example in Figure 7 to illustrate how saSLP combines short reductions to exploit the host's parallelism and register capacity.

```

void PR(std::vector<int> neighbors[],
        double *weight, double *rank)
{
    for (int i = 0; i < m; ++i)
        for (int k : neighbors[i])
            sum += weight[k] * rank[k];
}

```

(a) C++ source from PageRank

```

mov    d0, #0.0
L0: ldr    x8, [x1] ; neighbor ①
     ldr    d1, [x2, x8, #3] ; w ②
     ldr    d3, [x3, x8, #3] ; r ③
     fmul   d1, d1, d3 ; w * r ④
     fadd   d5, d0, d1 ; sum ⑤
     ldr    x9, [x1, #4] ①
     add    x1, x1, #8
     ldr    d2, [x2, x9, #3] ②
     ldr    d4, [x3, x9, #3] ③
     cmp    x1, x0
     fmul   d2, d2, d4 ; w * r ④
     fadd   d0, d5, d2 ; sum ⑤
     b.lt   L0

```

(b) ARMv8 guest binary (unrolled)

```

Entry: load %x0 %x1 %x2 %x3
Loop:  %d0 = phi [0, %Entry] [%d0, %Loop]
① %x8 = load i64* %x1
② %d1 = load f64* [%x2 + (%x8 << 3)]
③ %d3 = load f64* [%x3 + (%x8 << 3)]
④ %d1 = fmul f64 %d1, %d3
⑤ %d5 = fadd f64 %d0, %d1
① %x9 = load i64* [%x1 + 4]
  %x1 = add i64 %x1, 8
② %d2 = load f64* [%x2 + (%x9 << 3)]
③ %d4 = load f64* [%x3 + (%x9 << 3)]
  %cc = icmp ult i64 %x1, %x0
④ %d2 = fmul f64 %d2, %d4
⑤ %d0 = fadd f64 %d5, %d2
  br il %cc, label %Loop, %Exit
Exit: store %x1 %x8-x9 %d0 %d5 %d1-d4 %cc

```

(c) Typically mapped virtual registers and IR

Chain

Short instr. Reduction

Long instr. Reduction

(d) Data dependency graph

Combining graph

Long instr. Reduction

(e) Combined reduction

Fig. 7. An example of data-irregular loop with reduction.

First, to combine short reductions, saSLP collects short values that (1) live across loop iterations (i.e., phi nodes in the input IR) and (2) are used by a sequence of associative instructions with the same opcode to compute the values for the next iteration (i.e., reduction chain). The phi nodes and their reduction chains form cycles in their data dependency graphs. For instance, saSLP collects the phi node and its reduction chain (i.e., ⑤ and ⑤) in Figure 7(c). The phi node passes the reduction value between loop iterations and the reduction chain accumulates the multiply results in the reduction value, resulting in the dashed-line cycle in Figure 7(d).

Second, saSLP uses the same graph construction algorithm introduced in Section 3.2 to construct a combining graph on the reduction chain. This is because instructions in a reduction chain usually have a higher probability of reducing a sequence of similar instructions. Starting from the operands of the reduction chain, the algorithm traverses along the use-define chains and then groups short instructions that can be combined to form long SIMD instructions. Once the combining graph is built, saSLP combines short instructions in the reduction chain to form a long SIMD reduction and widens the phi node as well. In the PageRank example, the graph construction traversal starts from the multiply instructions (④, ④) in Figure 7(d) and traces along the use-define chains, resulting in the combining graph in Figure 7(e). Moreover, the 8B floating-point add instructions ⑤ and ⑤ in the reduction chain are combined and replaced with the long SIMD reduction ⑤⑤, which reduces a vector of two 8B floating-point numbers. The phi node is also widened to the width of ⑤⑤, as shown in Figure 7(e).

Finally, saSLP needs to horizontally reduce the long SIMD reduction to compute the final short reduction value that flows out of the loop. Static compilers achieve this by reducing the lanes of the long SIMD reduction at the loop exit. For example, as shown in Figure 8(a), the compiler inserts a horizontal reduction at the loop exit to horizontally sum the two 8B floating-point numbers inside the output vector of long SIMD reduction ⑤⑤. Assuming that the trip count of the loop is

Fig. 8. Limitations of typical horizontal reduction and the saSLP solution.

n and ϕ_i denotes the multiply result of ϕ in the i th iteration, the value of long SIMD reduction ϕ is $\langle \sum_{i=1}^n \phi_i, \sum_{i=1}^n \phi_i \rangle$ after the execution of the loop. Therefore, the horizontal reduction in Figure 8(a) sums the two lanes and thus gets $\sum_{i=1}^n \phi_i + \phi_i$ in %d0, which is the final reduction value.

However, the typical horizontal reduction that static compilers adopt does not work in DBTs. This is because all guest registers updated in the reduction chain must be synchronized back to the guest state, but typical horizontal reduction loses some of the updated guest registers. As Figures 7(c) and 7(d) show, the reduction chain in the unrolled loop modifies two registers, %d5 and %d0, whose values must be reflected in the guest states in memory. Guest register %d0 contains the final reduction result, and %d5 should accumulate the multiply results of ϕ and ϕ for n iterations except ϕ in the last iteration. That is, the final value of %d5 should be $\sum_{i=1}^{n-1} (\phi_i + \phi_i) + \phi_n$, which is less than the final reduction value %d0 by ϕ_n (Figure 8(c)). The typical horizontal reduction can resolve only the final result (e.g., %d0 but not %d5 in Figure 8(a)) and thus causes missing guest states.

To solve this problem, saSLP horizontally reduces the widened phi node instead of the long SIMD reduction and then sequentially reduces the reduction chain in the last iteration. Since all instructions in the reduction chain are sequentially executed, all guest registers updated in the chain are preserved. In the PageRank example shown in Figure 8(b), saSLP first horizontally reduces the widened phi node that holds $\langle \sum_{i=1}^{n-1} \phi_i, \sum_{i=1}^{n-1} \phi_i \rangle$, which is the accumulated result for $n - 1$ iterations (except the last iteration). The resulting short phi node at the loop exit contains $\sum_{i=1}^{n-1} \phi_i + \phi_i$. Next, saSLP unpacks the root of combining graph (i.e., ϕ) into registers %d1 and %d2, whose value is ϕ_n and ϕ_n , respectively. The reduction chain in Figure 8(b) then sums the short phi node, %d1 and %d2, to compute the final values of %d0 and %d5 for guest state synchronization. The value of each node in Figure 8(b) is shown in Figure 8(c). As a result, no guest state would be lost.

To ensure that combining reductions is profitable, saSLP uses the spill-aware cost model introduced in Section 3.4 to compute the performance costs. The additional costs of the horizontal reduction and the reduction chain at loop exit are added to the long SIMD cost c_L . Whether saSLP will combine the reductions or not is then determined by the same Equation (1).

3.6 Combining Indirect Loads to Form Gather Loads

To vectorize data-irregular loops that cannot be vectorized on the guest owing to the indirect array loads, saSLP combines the indirect loads and exploits the host's SIMD gather instructions.

Fig. 9. Overview of combining indirect loads and parsing gather pointer expressions.

This is a significant issue because the innermost loops in many important applications, such as graph algorithms, sparse matrix kernels, and scientific stencils, contain “indirectly load and then accumulate” pattern and are read-only. Therefore, saSLP combines indirect loads only in read-only loops so that a runtime aliasing check is not required. We use the same PageRank example to elaborate on why and how saSLP combines the indirect array loads.

Recall that the innermost loop of PageRank accumulates the weighted ranks of the neighboring nodes of a specific node. As Figures 7(a) and 9(a) show, the loads for weights and ranks (i.e., ② and ③) are indirect loads based on an index k loaded from the neighbors array (i.e., ①). Therefore, ARMv8 NEON compiler cannot vectorize the loop since NEON does not support gather loads, leaving data parallelism in the loop unexploited. However, the x86 AVX2 host supports gather instructions, making vectorizing the loop possible. As designed for vectorizing indirect array loads, AVX2 gather instructions take an array base pointer ($Base$), a vector of array indices (Idx), and the size of array element ($ESize$) to load memory location $Base + Idx[i] \times ESize$ to the i th lane of the output registers. Thus, static compilers can directly map indirect array loads to the instructions.

Without the explicit semantics of indirect array loads in source code, saSLP must analyze the data-dependency graphs of load pointers in the input IR to find the $Base$, Idx , and $ESize$. As Figure 9(b) shows, saSLP first locates the three expressions $Base$, Idx , and $ESize$ in the data-dependency graphs of the load pointers. Because the $Base$ expression is shared among the indirect array loads that could form a gather load, saSLP locates the $Base$ expression by finding a common operand of the add instructions in Figure 9(b). If the common operand exists, then the other operands of the add instructions should be the distances from $Base$ to the memory locations accessed by the indirect loads (i.e., $Idx \times ESize$). Therefore, saSLP then locates the $ESize$ expression by searching a constant expression shared by the multiply or left-shift instructions in Figure 9(b), and the other operands of the multiply or left-shift instructions are identified as the Idx expressions. Take PageRank as an example; saSLP analyzes the data-dependency graphs of the indirect loads ② and ③ in Figure 10(a) to locate the three expressions. First, saSLP discovers that $\%x2$ is shared by the two add instructions used by ② and ③ and, thus, deduces that $Base = \%x2$. Second, saSLP analyzes the other (i.e., right-hand side) operands of the add instructions and finds out that the constant 3 is shared by the two left-shift instructions in Figure 10(a). Therefore, saSLP concludes that $ESize = 2^3 = 8B$, which is the element size of the weight array. Last, ① and ④, the other (i.e., left-hand side) operands of the two left-shift instructions, are identified as Idx , as shown in Figure 10(a).

For data dependency graphs where saSLP can locate all of the expressions $Base$, Idx , and $ESize$, saSLP combines the indirect loads to form a gather load, as shown in Figure 9(c). The $Base$ expression is casted into a pointer of type T , whose size is $ESize$. Moreover, a new *get element pointer*

Fig. 10. An example of combining indirect array loads.

(GEP) instruction is added to compute the memory locations that would be gathered. The GEP instruction takes a pointer *Base* and a vector *Idx* and then outputs a vector of pointers pointing to $Base + Idx \times ESize$, where $ESize = sizeof(T)$ and T is the data type pointed by *Base*. As a result, the memory locations accessed by the gather load in Figure 9(c) are identical to the ones in Figure 9(b). On the other hand, if saSLP cannot locate expressions *Base*, *Idx*, and *ESize*, then the indirect array loads are non-combinable and packed into a long SIMD vector, as mentioned in Section 3.2.

In the PageRank example, because all three expressions are found, saSLP combines the two indirect loads ② and ② and forms an SIMD gather ② ② in Figure 10(b). In addition, $\%x2$ is casted to a pointer of 8B floating point number. The pointer is then used by the GEP instruction to compute the vector of gathered memory locations, whose value is $\langle \%x2 + ① \times 8, \%x2 + ① \times 8 \rangle$. Therefore, the gather instruction ② ② would load the same memory locations as the original indirect loads ② and ②, as shown in Figure 10(c). Similarly, the other indirect array loads ③ and ③ in Figure 10(a) are also combined. In addition to combining the indirect array loads, saSLP further explores more data parallelism by combining instructions in the *Idx* expression. For example, the load instructions of the neighbors indices array (i.e., ① and ①) are also combined in Figure 10(b) because they are contiguous according to the input IR and Figure 10(c).

To estimate the profits of combining indirect array loads, the performance costs of the original indirect loads and the new gather instructions are also counted in the short cost c_s and the long SIMD cost c_L , respectively. The spill-aware cost model mentioned in Section 3.4 then decides whether to combine the short instructions in the whole combining graph (including the indirect array loads) based on the same Equation (1).

3.7 Limitations of saSLP

When the input IR contains complicated structures, such as conditional branches, function calls, and SIMD permutation instructions, saSLP vectorizes the loop partially. In this case, the combining graph contains only the use-def chain from the seeds to the complicated instructions and the combining graph is the resulting scope for the saSLP vectorization. With the optimizations to eliminate complicated structures (e.g., if-conversion and function inlining), the whole loop would be vectorized by saSLP if there is no complicated structure left in the loop.

4 EVALUATION

We implemented saSLP in HQEMU (Hong et al. 2012), a retargetable cross-ISA DBT based on the LLVM 6.0 JIT compiler. We evaluated saSLP with translations from a guest ISA, ARMv8 NEON with

Fig. 11. Performance of NEON to AVX2 translation on Skylake.

32 16B SIMD registers, to two host ISAs, x86 AVX2 and AVX512. AVX2 and AVX-512 have 16 32B and 32 64B SIMD registers, respectively. The host systems for AVX2 and AVX-512 translation are Intel Core i7-6700 (Skylake, SKL) at 3.4GHz with 64GB of RAM and Intel Xeon Phi 7210 (Knights Landing, KNL) at 1.3GHz with 109GB of RAM, respectively.

The evaluation of saSLP consists of two parts. In the first part, we demonstrate and discuss the performance of saSLP with short NEON vector to long AVX vector transformation. All benchmarks used in the first part are data regular and thus are vectorizable on ARMv8 NEON. In the second part, we show that saSLP can further transform ARMv8 scalar to long AVX vector across a number of data-irregular benchmarks that are not vectorizable on ARMv8 NEON. All benchmarks used in the second part contain indirect array loads and loop reductions.

NEON Vector to AVX Vector Transformation

Fifteen benchmarks were selected from various benchmark suites for the evaluation: Livermore Loops (LL), NAS Parallel Bench (NPB) (Bailey et al. 1991), SciMark 2 (SM2), Test Suite for Vectorizing Compilers (TSVC) (Maleki et al. 2011), LINPACK (LP), jpeg (JPG) and OpenCV (CV). All benchmarks were compiled using AArch64 GCC 5.4.0 with flags “-O3 -ftree-vectorize -funroll-loops -ffast-math -static” to let the guest compiler automatically vectorize and unroll loops according to the NEON registers. For comparison, we used the translation without saSLP as our performance baseline, where NEON vectors were translated to AVX-128 vectors that have the same SIMD width.

ARMv8 Scalar to AVX Vector Transformation

In addition to short-vector to long-vector transformation, we also selected eight benchmarks from various applications and benchmark suites to evaluate saSLP with scalar to vector transformation. The selected benchmarks included PageRank (PR), High-Performance Conjugate Gradients (HPCG) (Dongarra et al. 2015), SM, TSVC and stochastic gradient descent (SGD). All benchmarks were compiled using the same ARMv8 guest compiler and flags as the short-vector to long-vector transformation. The compiler failed to vectorize the kernels in the eight data-irregular benchmarks since ARMv8 does not support SIMD gather instructions. As a result, the kernel loops were unrolled only according to the number of NEON registers. For comparison, we used the translation without saSLP as our performance baseline, where ARMv8 scalars were translated to x86 scalars.

4.1 Performance Results with NEON Vector to AVX2 Translation

Figure 11 shows the speedup of AVX2 translation with saSLP. By combining two NEON instructions, saSLP achieves a significant speedup of 1.6× on average compared to that without saSLP. Most of the benchmarks achieve more than 40% performance improvement with saSLP. In particular, the benchmarks *LL-hydro*, *LL-first_diff*, *LL-mat_mul*, *TSVC-vpvt*, and *CV-hc_sobel* achieve

Fig. 12. Overheads of JIT and runtime aliasing checks (AVX2 Skylake).

Fig. 13. Spilling load dynamic instruction ratio (AVX2 Skylake).

significant speedups over 1.8 $\times$. This is because their loops have good access patterns and locality, and contain a high ratio of SIMD computation. In contrast, *NPB-mg_psinv* and *NPB-mg_resid* achieve an average speedup of around 1.2 $\times$, which is relatively low. Profiling results indicate that *NPB-mg_psinv* and *NPB-mg_resid* have 17% of the total cycles with split loads (i.e., fetching two cache lines because the load crosses cache line boundary), which degrades the performance (Intel Corp. 2018a). Moreover, the working set of these two benchmarks is larger than the L3 cache size, leading to 30% cycles on L3 miss stalls. Since these memory overheads cannot be reduced by the improved parallelism, there is little room for saSLP to enhance the performance.

The overheads of JIT compilation and runtime aliasing checks are measured as the ratios to the total execution time in Figure 12. Since all loops in the benchmark set are translated once but iterate many times, the JIT overhead is amortized to 3%, on average. Moreover, our DBT uses another thread for LLVM JIT so that the overhead can often be hidden. The results also show that runtime aliasing check incurs a negligible overhead of 2.95%, on average, because the checking code is executed only once before entering the loop. As a result, the overhead is amortized. One exception is *JPG-8x8fDCT*, in which SIMD loops have relatively small trip counts; consequently, the overhead of runtime aliasing checks cannot be fully amortized.

To evaluate the impacts from the automatic unrolling factors made by the guest compiler, we force the guest compiler to compile benchmarks with smaller unrolling factors, which results in no register spilling on the AVX2 host. As shown in Figure 11, the overall speedup decreases from 1.6 $\times$ to 1.47 $\times$ because there is no spilling to eliminate. However, even in this case, saSLP can still improve the performance by exploiting the AVX2's parallelism.

4.2 Register Spilling with NEON Vector to AVX2 Translation

Figures 13 and 14 show the dynamic instruction ratios of register spilling loads (i.e., filling) and stores (i.e., spilling) against the total executed instruction counts. On average, saSLP eliminates

Fig. 14. Spilling store dynamic instruction ratio (AVX2 Skylake).

67% and 97% of spilling loads and stores, respectively, by combining two 16B NEON registers and storing them in a 32B YMM register on the AVX2 host.

For the spilling loads in Figure 13, benchmarks such as *LL-hydro*, *LL-first_diff*, *NPB-mg_reside*, *TSVC-vpvt*, *LP-daxpy*, *CV-gauss_2D*, and *CV-hc_sobel* show a significant reduction in spilling loads when saSLP is applied. On average, these benchmarks achieve 87% reduction because the number of interfered live ranges is significantly reduced by combining them. On the other hand, *LL-int_pred* and *JPG-8x8fDCT* do not achieve a significant reduction because most of the NEON registers in the loops are duplicated and fill up the YMM registers rather than being combined with each other. Furthermore, for the spilling stores in Figure 14, nearly all of the benchmarks achieve significant reductions, exceeding 70%. Almost all live ranges that flow through the loops for guest state synchronization are combined and stored in the YMM registers. Therefore, the spilling stores in the loops are eliminated as a consequence of the decrement of interfering live ranges.

The reduction in register spilling also impacts the performance depending on the load/store pressure of the benchmarks. Therefore, we profile the dispatched load/store micro-ops and memory stall cycles using the hardware performance counters. In Figures 13 and 14, the dashed lines (right *y*-axis) indicate the ratios of the execution cycles with dispatched load micro-ops over the total execution cycles and the ratios of the store micro-ops, respectively. The results in Figures 11, 13, and 14 show that the benchmarks with relatively high loads and store pressure *and* significant reduction in register spilling, such as *LL-hydro* and *LL-first_diff*, achieve speedups exceeding 2× (i.e., the theoretical bound of speedup from improved parallelism). The benchmarks with high store pressure *and* significant reductions in spilling stores, such as *LL-mat_mul*, *TSVC-expand*, and *TSVC-vpvt*, also achieve significant speedups, ranging from 1.71× to 1.83×.

4.3 Performance Results with NEON Vector to AVX-512 Translation

Figure 15 shows the speedup of AVX-512 translation with saSLP, where four or two NEON instructions are combined. On average, saSLP achieves a speedup of 2.3× compared to the translation without saSLP. All benchmarks from the *LL*, *TSVC*, and *CV* suites achieve a significant speedup of 2.9×, on average. In contrast, *NPB-mg_psinv*, *NPB-mg_resid*, *SM2-LU_factor*, and *JPG-8x8fDCT* have an average speedup of 1.5×. *NPB-mg_psinv* and *NPB-mg_resid* suffer from large working sets and excessive split loads. Moreover, the performance degradation caused by the split loads is more severe on AVX-512 because the SIMD width is the same as the 64B cache line size. If the start addresses of the AVX-512 loads are not aligned to 64B, then all subsequent loads will always cross cache line boundaries and incur a significant penalty (Intel Corp. 2018a). The profile data show that both benchmarks have 37% of split loads, and a similar phenomenon is also observed in *SM2-LU_factor*. For *JPG-8x8fDCT*, the speedup is similar to that of the translation to AVX2 because saSLP has only two 16B NEON instructions to combine in the loops.

Fig. 15. Performance of NEON to AVX-512 translation (Knights Landing).

Fig. 16. Overheads of JIT and runtime aliasing checks (AVX-512 Knights Landing).

Fig. 17. Spilling load dynamic instruction ratio (AVX-512 Knights Landing).

As shown in Figure 16, the overheads of JIT compilation and runtime aliasing checks are similar to those in the AVX2 translation. Because of the amortization of overheads, the JIT compilation and runtime aliasing checks incur only negligible costs of 1.65% and 4.28% of the total execution time, on average, respectively.

Figures 17 and 18 show the dynamic instruction ratio of register spilling loads and stores with the AVX-512 translation, respectively. On average, saSLP eliminates 95% and 99% of spilling loads and stores, respectively. Although AVX-512 supports 32 64B ZMM registers, accessing the extra 16 XMM16-31/YMM16-31 sub-registers requires AVX-512VL extension, which is not supported on our host. Therefore, the baseline that translates to XMM width can only use 16 XMM registers, while saSLP can exploit all 32 ZMM registers by combining with the ZMM width. As a consequence, saSLP further reduces spilling with the extra ZMM registers. For example, the spilling loads caused by the duplicated and packed NEON registers in *LL-int_pred* (Figure 13) can be eliminated by storing these duplicated NEON registers in the extra ZMM registers, resulting in a 98% reduction in the spilling loads (as shown in Figure 17). For the spilling store in Figure 18, benchmarks

Fig. 18. Spilling store dynamic instruction ratio (AVX-512 Knights Landing).

such as *LL-hydro*, *LL-diff_pred*, *LL-frag2D*, and *NPB-mg_psin* also benefit from the improved SIMD register capacity after the saSLP transformation. On average, there is 82% extra reduction in the spilling stores compared to the AVX2 translation.

Our AVX-512 host machine does not support the hardware counters that count the cycles with dispatched load/store micro-ops to estimate the load/store pressure. Therefore, we estimate the load/store pressure by calculating the ratios of accumulated memory latency (from the event counts and latencies of L1, L2 data cache and DRAM) over the total execution cycles (Jeffers et al. 2016). Figures 15, 17, and 18 show that, similar to the AVX2 translation, benchmarks with a high load and store pressure *and* significant reduction in register spilling achieve significant speedup. Specifically, *LL-hydro*, *LL-first_diff*, and *CV-hc_soble* achieve 2.97 $\times$, 2.91 $\times$, and 2.95 $\times$ speedup, respectively. In addition, benchmarks with high store pressure *and* significant reduction in spilling stores, such as *LL-mat_mul*, *TSVC-expand*, and *TSVC-vpvt*, achieve significant speedups, ranging from 2.6 $\times$ to 3.54 $\times$.

To summarize, in the first part of the experiments, we evaluated saSLP with the translations from the short SIMD ISA, ARMv8 NEON, to two long SIMD ISAs, x86 AVX2 and AVX-512. The benchmark results show that saSLP can effectively exploit the improved parallelism and register capacity of the host. Moreover, it can achieve a significant reduction in the register spilling overhead, showing a speedup of 1.6 $\times$ and 2.3 $\times$ for the AVX2 and AVX-512 hosts, respectively.

4.4 Performance Results with ARMv8 Scalar to AVX2 Translation

In addition to combining NEON short vectors to form AVX2 long vectors, saSLP also uses AVX gather instructions to vectorize loops containing reductions and indirect array loads, which are not vectorizable on ARMv8 NEON. We evaluated saSLP with such scalar to SIMD vector transformation across a number of data-irregular applications and benchmarks.

Figure 19 shows the speedup of ARMv8 scalar to AVX2 translation with saSLP. By vectorizing the scalar reduction chains and packing the scalars from indirect array loads into SIMD vectors, saSLP achieves a speedup of 1.75 $\times$, on average, compared to that without saSLP. Moreover, when using the AVX2 gather instructions to vectorize the indirect array loads, saSLP further improves the average speedup to 1.92 $\times$. In particular, benchmarks *SM2-SpMM*, *TSVC-s4115*, and *TSVC-s4116* significantly benefit from the AVX2 gather instructions, resulting in an extra speedup of 13%, on average. All of the single-precision benchmarks achieve higher speedups with saSLP than the double-precision benchmarks (i.e., 3.73 $\times$ vs. 1.29 $\times$). This is because AVX2 single-precision instructions process two times more vector elements in parallel than the double-precision instructions. Furthermore, single-precision numbers require less memory bandwidth and capacity.

Although saSLP significantly improves the performance of the benchmarks with SIMD vectorization, the speedup that can be accomplished is still limited by the following factors. (1) Data

Fig. 19. Performance of ARMv8 scalar to AVX-256 translation on Skylake.

Fig. 20. Composition of stall cycles caused by load instructions.

access pattern and locality of the benchmarks: if the memory access pattern is irregular (e.g., indirect loads) or the working set cannot fit the cache, it would be difficult to maintain the theoretical SIMD computation throughput due to the latency of cache miss. (2) According to Amdahl's law, the speedup is limited by the ratio of non-SIMD code. For example, *PR-small* achieves 38% higher speedup than *PR-large*, which processes much larger network graphs. This is because *PR-large* has higher ratio of stall cycles for waiting data from DRAM and last-level cache (LLC) compared to *PR-small*, as shown in Figure 20. Moreover, *PR-small* has much higher ratio of stall cycles on L1 cache hit, resulting in less memory overheads and greater performance gain. Similarly, *HPCG-SpMV* and *HPCG-SymGS* also suffer from high ratios of stall cycles on DRAM and, thus, achieve less speedup. Furthermore, since the kernels in the *HPCG* suite are 27-point stencils and the vectorized kernels have low trip counts, non-SIMD code becomes the bottleneck and impedes the speedup.

On the other hand, single-precision benchmarks *SM2-SpMM*, *TSVC-s4115*, and *TSVC-s4116* have most of their stall cycles waiting for L1 and L2 cache hits instead of LLC and DRAM (Figure 20), resulting in a significant speedup of 3.73 $\times$, on average. This is because these benchmarks have a smaller working set and less divergence in the indirect load addresses.

Figure 19 shows that the benefit of exploiting AVX2 gather instructions instead of using software packing (i.e., inserting indirectly loaded scalars into a vector) is more significant among single-precision benchmarks. Since the AVX2 gather instructions have high latency and are implemented with complex micro-ops, the overheads of gather instructions need to be amortized among the gathered elements for good performance (Hirsh and Gideon 2017). In fact, the performance of gathering four double-precision numbers with AVX2 gather instructions is on par with software packing on Skylake micro-architecture, while gathering eight single-precision numbers with hardware is up to 70% faster than software packing (Intel Corp. 2018a). Therefore, by combining more elements to amortize the overheads of AVX2 gather instructions, saSLP achieves higher extra speedup of exploiting hardware gathering in the single-precision benchmarks.

Fig. 21. Performance of ARMv8 scalar to AVX-512 translation on Knights Landing.

Table 1. Measurement of the Loops and Performance Speedups of the Benchmark Suites

	LL	TSVC	NPB-mg	SGD
# Loops	21	28	33	36
# Loops filtered	10	9	26	32
Speedup	1.67	1.88	1.09	1.06

4.5 Performance Results with ARMv8 Scalar to AVX-512 Translation

Figure 21 shows the speedup of ARMv8 scalar to AVX-512 translation with saSLP. On average, saSLP achieves an average speedup of 1.74 $\times$ by vectorizing the kernels with AVX-512 gather instructions. Compared with the AVX2 translation, the benefit of using hardware gather instructions is more significant on AVX-512. By replacing software packing with AVX-512 hardware gathering, saSLP boosts the performance by 60%, on average. In particular, *SGD-predict* and *TSVC* achieve extra speedup over 2 $\times$. This is because AVX-512 can gather more elements to amortize the cost, and software packing has higher latency on Knights Landing (Fog 2018; Intel Corp. 2018a).

Though AVX-512 has wider vectors than AVX2, the overall performance gain on Knights Landing (AVX-512) is less significant than that on Skylake (AVX2). The speedup achieved on Knights Landing (KNL) is limited by the high latency of indirect loads. Since KNL does not have LLC, indirect loads that miss L2 would incur DRAM latency (≈ 200 cycles). Moreover, the out-of-order resources on KNL is much less than Skylake (Fog 2018; Intel Corp. 2018a), making it difficult for KNL to hide the high latency. Even so, saSLP can still achieve speedup ranging from 1.9 $\times$ to 4.2 $\times$ in benchmarks with better locality, such as *SGD-predict*, *SM2-SpMM*, and *TSVC*.

4.6 Evaluation of Spill-Aware Cost Model

Table 1 lists the total number of loops in the benchmarks and the number of loops that are filtered out by our spill-aware cost model. In this experiment, the kernels from the same benchmark suite are grouped in a whole program. We conduct the measurement with ARMv8 to AVX2 translation on the Intel Skylake platform. The optimizations for loop reductions and indirect loads are enabled. As the result shows, many loops in *LL* and *TSVC* pass the spill-aware cost model evaluation and are vectorized, while many loops in *NPB-mg* and *SGD* are filtered out. The successfully vectorized loops are from the computational kernels and several array initialization routines. The other loops fail in the cost evaluation because their combining graphs are too small (i.e., insufficient data parallelism) or significant overhead could be incurred due to excessive data reorganization.

Since *LL* and *TSVC* have more vectorized loops, they have higher speedups than *NPB-mg* and *SGD*, as shown in Table 1. The speedups do not achieve the theoretical value, mainly because,

according to Amdahl's law, the speedup is bounded by the ratio of the non-SIMD portions, where time is spent for benchmark data initialization and library routines.

5 RELATED WORK

Auto-vectorization approaches are usually classified into two types: loop-based vectorization and basic-block vectorization. Among the latter, Superword Level Parallelism (SLP) (Larsen and Amarasinghe 2000) is the most well-known algorithm. It can combine multiple isomorphic statements into SIMD operations inside a straight-line code. A number of techniques have been developed to improve SLP, such as dealing with the presence of control flow (Shin et al. 2005), padding with non-isomorphic statements (Porpodas et al. 2015), and partial SIMD parallelism (Zhou and Xue 2016a). In this article, the proposed saSLP extends SLP to resolve the asymmetric SIMD problems in cross-ISA DBTs. Our extensions include: (1) re-vectorizing scalar/vector to longer vector; (2) runtime aliasing check; (3) the spill-aware cost model; and (4) vectorizing indirect loads with SIMD gather instructions.

Some auto-vectorizers combine loop-based and SLP vectorization to extract both inter- and intra-iteration parallelism at the same time (Rosen et al. 2007; Kim and Han 2012; Zhou and Xue 2016b). Rosen et al. (2007) and Zhou and Xue (2016b) target data-regular loops, while Kim and Han (2012) can vectorize irregular code. Their work focuses on reducing the data reorganization overhead incurred in vectorizing loops. In contrast, saSLP resolves the asymmetric SIMD capability problems in cross-ISA DBTs with a spill-aware cost model to determine whether combining the guest instructions would be profitable.

In contrast to classical SIMD architectures, the architecture of scalable vectors, such as the ARM Scalable Vector Extension (SVE) (ARM Ltd. 2017) and RISC-V vector extension (RISC-V Foundation 2016), aims to provide binaries that can be configured to varying vector widths without re-compilation. In contrast to the scalable vector, which requires special hardware support, our method is a pure software solution that can run on existing commodity processors and is transparent to existing application binaries.

DBT is a core technology used to execute an application binary of one ISA on a host machine with a different ISA. DBT has been used to achieve many different goals. Dynamic optimization systems (Bala et al. 2000; Lu et al. 2004; Wang et al. 2007) apply runtime binary re-translation to improve performance; binary instrumentation systems, such as Pin (Luk et al. 2005) and Valgrind (Nethercote and Seward 2007), provide features of application analysis and debugging. DBT is also widely used for application migration across ISAs (Baraz et al. 2003; Bellard 2005; Chernoff et al. 1998). For example, the cross-ISA DBT systems, such as Digital FX!32 (Chernoff et al. 1998), IA-32 EL (Baraz et al. 2003), and Transmeta CMS (Dehnert et al. 2003), enabled IA-32 applications to be executed on Alpha, Itanium, and VLIW machines in the past.

With the advances in SIMD architectures, several DBT systems that attempt to exploit SIMD extensions have been proposed in the literature. However, they have very different objectives and approaches in their designs. The most closely related works are Hallou et al. (2016), Hallou et al. (2015), and Hong et al. (2016), which present vectorization schemes from short SIMD to long SIMD for loops. Hallou et al. designed their work in a same-ISA DBT for the x86 architecture, while Hong et al. proposed a cross-ISA solution. Their approaches use loop-based vectorization to improve the parallelism. However, they still map one guest register to one host register, which cannot eliminate register spilling. In contrast, on the basis of the SLP algorithm, saSLP can combine multiple short guest registers and solve the register spilling problem as a result. Moreover, saSLP can exploit the host's advanced SIMD instructions to vectorize data-irregular loops.

Pajuelo et al. (2002) proposed a binary auto-vectorization framework from scalars to vectors. Their system requires new hardware to execute speculatively vectorized code and falls back to

scalar code when an invalid vectorization is detected by the hardware. Liquid SIMD (Clark et al. 2007) proposed a general SIMD representation, which can run on Liquid SIMD-enabled hardware. In contrast, our saSLP is a pure software solution running on the existing hardware. Vapor SIMD (Nuzman et al. 2011) proposed split vectorization, where offline compilers conduct vectorization of machine-independent parts and runtime JITs resolve the final vectorization based on the host SIMD ISA. In contrast to Vapor SIMD, our saSLP works for existing application binaries.

QEMU (Bellard 2005) emulates a guest SIMD instruction with a sequence of scalar instructions instead of leveraging host SIMD capability for portability. Other works (Fu et al. 2015; Li et al. 2006; Michel et al. 2011) focus on finding efficient translations of SIMD binaries between different SIMD ISAs. In their DBTs, guest SIMD instructions are translated to equivalent host SIMD instructions that operate on the same number of elements even if the host can provide more parallelism. In contrast, saSLP combines short guest instructions to form longer SIMD instructions so that the host's parallelism is fully used.

6 CONCLUDING REMARKS

The diversity of modern SIMD architectures in registers and advanced instructions has become a critical issue when applying DBTs for cross-ISA migration of applications. This issue will become more common and severe with the ongoing evolution of SIMD architectures. Since translating from longer to shorter SIMD is an easier problem that splitting vectors can solve, this article addresses the more complex optimization problem of leveraging longer SIMD on the host.

We propose a novel cross-ISA DBT technique, called spill-aware SLP (saSLP), which can combine multiple short ARMv8 instructions and registers to fully use the x86 AVX host's parallelism and register capacity. We also propose the spill-aware cost model and runtime aliasing check to prevent anticipated slowdown and violation of vectorization legality. Furthermore, saSLP exploits the x86 AVX gather instructions to vectorize data-irregular loops, which cannot be vectorized on ARMv8 NEON. We also present a new algorithm to vectorize loop reductions in DBTs, where the typical methods used in static compilers do not work. The evaluation results of saSLP demonstrate that the algorithm can significantly improve the execution efficiency and effectively reduce the register spilling across a range of benchmarks. Moreover, the proposed technique is also applicable to static binary translation (SBT) and can be further extended to support more guest and host SIMD ISAs since saSLP is built on top of the LLVM machine-independent IR layer.

REFERENCES

- ARM Ltd. 2015. *ARM Cortex-A Series Programmer's Guide for ARMv8-A*.
- ARM Ltd. 2017. *ARM Architecture Reference Manual Supplement: The Scalable Vector Extension (SVE), for ARMv8-A*.
- David H. Bailey, Eric Barszcz, John T. Barton, D. S. Browning, Robert L. Carter, Leonardo Dagum, Rod A. Fatoohi, Paul O. Frederickson, T. A. Lasinski, Robert Schreiber, Horst D. Simon, V. Venkatakrishnan, and Sisira Weeratunga. 1991. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications* 5, 3 (1991), 63–73.
- Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'00)*. ACM, 1–12.
- Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigal Zemach. 2003. IA-32 execution layer: A two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO'03)*. ACM/IEEE Computer Society, 191–204.
- Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*. USENIX, 41–46.
- Anton Chernoff, Mark A. Herdeg, Raymond J. Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. 1998. FX!32—a profile-directed binary translator. *IEEE Micro* 18, 2 (1998), 56–64.
- Nathan Clark, Amir Hormati, Sami Yehia, Scott A. Mahlke, and Krisztián Flautner. 2007. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In *Proceedings of the International Conference on High-Performance Computer Architecture (HPCA'07)*. IEEE Computer Society, 216–227.

- James C. Dehnert, Brian Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 2003. The transmeta code morphing—software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*. IEEE Computer Society, 15–24.
- Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable reverse engineering with PANDA. In *Proceedings of the Program Protection and Reverse Engineering Workshop (PPREW@ACSAC'15)*. ACM, 4:1–4:11.
- Jack Dongarra, Michael A. Heroux, and Piotr Luszczek. 2015. *HPCG Benchmark: A New Metric for Ranking High Performance Computing Systems*. Technical Report. Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville.
- Agner Fog. 2018. *Lists of Instruction Latencies, Throughputs and Micro-operation Breakdowns for Intel, AMD and VIA CPUs*. https://www.agner.org/optimize/instruction_tables.pdf.
- Sheng-Yu Fu, Ding-Yong Hong, Jan-Jan Wu, Pangfeng Liu, and Wei-Chung Hsu. 2015. SIMD code translation in an enhanced HQEMU. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS'15)*. IEEE Computer Society, 507–514.
- Nabil Hallou, Erven Rohou, and Philippe Clauss. 2016. Runtime vectorization transformations of binary code. *International Journal of Parallel Programming* 45, 6 (2016), 1536–1565.
- Nabil Hallou, Erven Rohou, Philippe Clauss, and Alain Ketterlin. 2015. Dynamic re-vectorization of binary code. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'15)*. IEEE Computer Society, 228–237.
- Israel Hirsh and S. Gideon. 2017. *Intel Architecture Code Analyzer User's Guide*.
- Ding-Yong Hong, Sheng-Yu Fu, Yu-Ping Liu, Jan-Jan Wu, and Wei-Chung Hsu. 2016. Exploiting longer SIMD lanes in dynamic binary translation. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS'16)*. IEEE Computer Society, 853–860.
- Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. 2012. HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'12)*. ACM, 104–113.
- Sungpack Hong, Tayo Oguntobi, and Kunle Olukotun. 2011. Efficient parallel graph exploration on multi-core CPU and GPU. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Lawrence Rauchwerger and Vivek Sarkar (Eds.). IEEE Computer Society, 78–88.
- Xin Huo, Vignesh T. Ravi, Wenjing Ma, and Gagan Agrawal. 2011. An execution strategy and optimized runtime support for parallelizing irregular reductions on modern GPUs. In *Proceedings of International Conference on Supercomputing, (ICS'11)*. ACM, 2–11.
- Intel Corp. 2018a. *Intel 64 and IA-32 Architectures Optimization Reference Manual*.
- Intel Corp. 2018b. *Intel 64 and IA-32 Architectures Software Developer's Manual*.
- J. Jeffers, J. Reinders, and A. Sodani. 2016. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Elsevier Science.
- Seonggun Kim and Hwansoo Han. 2012. Efficient SIMD code generation for irregular kernels. In *Proceedings of International Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM, 55–64.
- Samuel Larsen and Saman P. Amarasinghe. 2000. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'12)*. ACM, 145–156.
- Chris Lattner and Vikram S. Adve. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. IEEE Computer Society, 75–88.
- 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, 269–280.
- Yu-Ping Liu, Ding-Yong Hong, Jan-Jan Wu, Sheng-Yu Fu, and Wei-Chung Hsu. 2017. Exploiting asymmetric SIMD register configurations in ARM-to-x86 dynamic binary translation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'17)*. IEEE Computer Society, 343–355.
- Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei-Chung Hsu. 2004. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism* 6 (2004), 1–24.
- Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, 190–200.
- Saeed Maleki, Yaoqing Gao, María Jesús Garzarán, Tommy Wong, and David A. Padua. 2011. An evaluation of vectorizing compilers. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. IEEE Computer Society, 372–382.

- Luc Michel, Nicolas Fournel, and Frédéric Pétrot. 2011. Speeding-up SIMD instructions dynamic binary translation in embedded processor simulation. In *Proceedings of the Design, Automation and Test in Europe (DATE'11)*. IEEE Computer Society, 277–280.
- Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, 89–100.
- 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 International Symposium on Code Generation and Optimization (CGO'11)*. IEEE Computer Society, 151–160.
- Alex Pajuelo, Antonio González, and Mateo Valero. 2002. Speculative dynamic vectorization. In *Proceedings of the International Symposium on Computer Architecture (ISCA'02)*. IEEE Computer Society, 271–280.
- Vasileios Porpodas, Alberto Magni, and Timothy M. Jones. 2015. PSLP: Padded SLP automatic vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'15)*. IEEE Computer Society, 190–201.
- RISC-V Foundation. 2016. *RISC-V Vector Extension Proposal*.
- Ira Rosen, Dorit Nuzman, and Ayal Zaks. 2007. Loop-aware SLP in GCC. In *Proceedings of the GCC Developers Summit*. Red Hat Inc., 131–142.
- Jaewook Shin, Mary W. Hall, and Jacqueline Chame. 2005. Superword-level parallelism in the presence of control flow. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)*. IEEE Computer Society, 165–175.
- Cheng Wang, Shiliang Hu, Ho-Seop Kim, Sreekumar R. Nair, Mauricio Breternitz Jr., Zhiwei Ying, and Youfeng Wu. 2007. StarDBT: An efficient multi-platform dynamic binary translation system. In *Proceedings of the Asia-Pacific Computer Systems Architecture Conference, Lecture Notes in Computer Science*, Vol. 4697. Springer, Berlin, 4–15.
- Hao Zhou and Jingling Xue. 2016a. A compiler approach for exploiting partial SIMD parallelism. *ACM Transactions on Architecture and Code Optimization* 13, 1 (2016), 11:1–11:26.
- Hao Zhou and Jingling Xue. 2016b. Exploiting mixed SIMD parallelism by reducing data reorganization overhead. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'16)*. ACM, 59–69.

Received July 2018; revised October 2018; accepted November 2018