Exploiting Longer SIMD Lanes in Dynamic Binary Translation

Ding-Yong Hong[1], Sheng-Yu Fu[2], Yu-Ping Liu[2], Jan-Jan Wu[1] and Wei-Chung Hsu[2]
[1]Institute of Information Science, Academia Sinica
Email: {dyhong,wuj}@iis.sinica.edu.tw
[2]Department of Computer Science and Information Engineering, National Taiwan University
Email: {d03922013,r04922005,hsuwc}@csie.ntu.edu.tw

***Abstract*—Recent trends in SIMD architecture have tended toward longer vector lengths and more enhanced SIMD features have been introduced in the newer vector instruction sets. However, legacy or proprietary applications compiled with short-SIMD ISA cannot benefit from the long-SIMD architecture, which supports improved parallelism and enhanced vector primitives, and thus only achieve a small fraction of potential peak performance. This paper presents a dynamic binary translation technique that enables short-SIMD binaries to exploit the benefits of the new SIMD architecture by rewriting short-SIMD loop code. We propose a general approach that translates loops consisting of short-SIMD instructions to machine-independent IR, conducts SIMD loop transformation/optimization at this IR level, and finally translates to long-SIMD instructions. Two solutions are presented to enforce SIMD load/store alignment, one for the problem caused by the binary translator's internal translation condition and one general approach using loop peeling optimization. The benchmark results show that an average speedup of 1.45X is achieved for NEON to AVX2 loop transformation.**

***Index Terms*—Dynamic binary translation, SIMD, vectorization, alignment, dynamic loop peeling.**

I. Introduction

Dynamic Binary Translation (DBT) is a common technique to execute an application binary of one guest ISA on a host machine with a different ISA. It operates directly on the binary, providing the powerful functionality to inspect and modify the runtime behavior of the applications without source code available. One important application of DBT is that it allows for cross-ISA migration of legacy or proprietary applications. For example, many Android applications are either compiled for the ARM ISA or include libraries in ARM native code in the APK package. Based on DBT technique, the widely used Android emulator can run such ARM executables on x86-based platforms. The emulation performance is the key to the success of a cross-ISA DBT. Hence, optimizing the translated code quality is important in the design of a cross-ISA DBT.

Single Instruction Multiple Data (SIMD) architectures are emerging as an essential extension to modern microprocessors, such as NEON for ARM, SSE/AVX for x86, and MSA for MIPS. The execution of a SIMD instruction simultaneously performs the same operation on multiple data, and is power- and execution-efficient. Recent trends in SIMD architecture development have also tended toward wider vector lengths. For instance, the Intel x86 architecture has gone through several generations: from 128-bit SIMD with SSE to 256-bit with AVX, to the most recent extension, AVX-512, which is supported in Intel Skylake Xeon platforms. In addition, more enhanced SIMD features are introduced in the newer SIMD ISA, e.g., fused multiply-and-add (FMA) and gather/scatter instructions. Many applications have enabled SIMD computations and achieved performance improvements of several orders of magnitude, such as multimedia processing and scientific computing [1]–[3]. We can expect that more applications utilizing SIMD ISA will be developed.

The importance of SIMD has increased the importance of supporting efficient translation of SIMD instructions in a DBT system. Several prior works have proposed solutions to support translation of SIMD instructions in cross-ISA DBTs. QEMU [4] and the Android emulator emulate a SIMD instruction by using a sequence of scalar instructions instead of leveraging the host SIMD instructions. Alternatively, [5]–[8] focus on finding effective mappings of SIMD registers and instructions between different guest and host SIMD architectures. In these DBT systems, guest SIMD instructions are translated to equivalent host SIMD instructions using *the same* number of lanes (e.g., ARM NEON instruction `vadd.i32` is translated to x86 SSE `paddl`, both are for four-integer vector addition).

Unfortunately, for legacy application binary to run on host architectures with longer SIMD lanes, these works still translate guest SIMD instructions to host SIMD instructions for the same SIMD width. As a result, these approaches cannot benefit from the improved parallelism provided by the longer SIMD lanes. Only a small fraction of the potential peak performance can be achieved for the applications.

This paper seeks to exploit full parallelism of the longer host SIMD lanes to accelerate the execution of guest SIMD instructions. We propose a DBT system that enables *short-SIMD* binaries to exploit the *long-SIMD* architecture by rewriting the short-SIMD binaries. We aim to transform a short-SIMD loop into an equivalent long-SIMD loop optimized with long-SIMD instructions. By increasing the number of SIMD elements processed in each loop iteration, the execution time can be shortened. This work focuses on rewriting *loops* that consist of short-SIMD instructions; the loops before and after the transformation are respectively denoted as short-SIMD and long-SIMD loops. We also improve the translated code quality by exploiting new SIMD features in our DBT.

Designing such loop transformation involves several issues when being applied to a longer vector. First, how will the loop control be managed? This includes the adjustment of the long-SIMD loop iteration and the situation if the number of short-SIMD loop iteration is not divisible by that of the long-

SIMD loop. Second, there is a legality issue where in that the short-SIMD loop is not always re-writable to the long-SIMD loop due to memory dependence conflicts. This issue is more challenging in a DBT than in a static compiler because a DBT usually translates small granularity binary segments. Based on the limited information from the input code segment, a DBT may not always be able to determine whether the transformation would violate the dependence at time of translation. Third, the SIMD architecture usually requires that data be aligned to certain boundary for efficient memory access, usually based on the vector length. This causes the problem that memory accesses aligned in the original binary may become misaligned when using long-SIMD instructions. Hence, how to enforce SIMD load/store alignment remains a challenging issue. The main contributions of this paper are as follows:

- We present the dynamic binary translation technique to rewrite short-SIMD code to long-SIMD instructions. We propose a general approach that translates the short-SIMD binary to machine-independent IR, conducts SIMD loop transformation at this IR level, and finally translates to host long-SIMD instructions.
- We identify the alignment problem in which aligned memory accesses can become misaligned when translated to long-SIMD instructions. Two solutions, one for the problem caused by the DBT internal translation condition and one general approach using loop peeling optimization, are presented to enforce SIMD load/store alignment.
- We implemented the proposed SIMD loop transformation in a cross-ISA DBT. The effectiveness of the approach is evaluated with translations from the short-SIMD ISA (NEON) to two long-SIMD ISAs (AVX2, AVX512). Benchmark results show that an average speedup of 1.45X can be achieved by doubling the vector length.

The rest of this paper is organized as follows. Section II presents the approach of short-SIMD to long-SIMD loop transformation. Section III describes the alignment problem and our solutions. Section IV reports the experimental results. Section V lists related work and Section VI concludes.

II. Short-to-Long SIMD Loop Transformation

The SIMD loop transformation is designed in HQEMU [9], a DBT that integrates QEMU and an LLVM JIT compiler. When a hot loop is detected, the loop binary is converted to QEMU IR which is then converted to LLVM IR for optimization. Since loops are usually good vectorization candidates for traditional compilers and our DBT also selects loops for LLVM optimization, this work focuses on *loops* as the SIMD transformation candidates. We perform the transformation on LLVM IR. The goal is to transform the IR of a short-SIMD loop to an equivalent long-SIMD loop representation. As a result, host long-SIMD instructions are emitted and advantages of the host SIMD architecture can be exploited.

A. An Example of Short-to-Long SIMD Loop Transformation

Figure 1 shows the transformation from a short-SIMD loop (ARM32 NEON) to a long-SIMD loop (x86-64 AVX2). Fig. 1(a) lists the loop source code. The source code is compiled by GCC to a scalar loop, a NEON vector loop and a runtime pointer check block in the binary. The assembly code of the NEON vector loop body is shown in Fig. 1(b). In this example, the base pointers of array x, y and z are respectively mapped to registers $\%r1$, $\%r2$ and $\%r3$. Four consecutive single-precision floating-point data elements are executed at a time with NEON. Fig. 1(c) shows the short-SIMD loop IR translated from the NEON loop binary by the DBT.

From the source code in Fig. 1(a), we can observe that the loop can also be vectorized with AVX2. Since AVX2 can simultaneously operate 8 float data elements, we would like to transform the short-SIMD loop to execute 8 elements at a time instead of 4 in the original binary. The AVX2 vector length is twice the width of the NEON vector. Thus, ideally we will double the number of vector elements of each IR vector type, modify the increasing value of the index from 16 to 32, and adjust the end iteration of vector loop. The finalized long-SIMD loop IR is shown in Fig. 1(d).

The example in Fig. 1 shows that several issues have to be solved for the transformation:

First, as the vector length is widened, the long-SIMD loop will compute more data elements in a single loop iteration. This implies that the number of loop iterations should be reduced. Moreover, there are situations in which the short-SIMD loop iterations cannot be completely consumed by the long-SIMD loop. For instance, the last index the long-SIMD loop can reach is 1024 instead of 1028 (i.e., 4096 instead of 4112 in byte units). The remaining iterations still need to be handled by the short-SIMD loop. This raises the issue of managing loop controls of both short- and long-SIMD loops.

Second, traditional compilers need to consider memory dependence when performing vectorization. Such legality issue also needs to be verified in our loop transformation. This issue is more challenging especially in a DBT compared with traditional compilers because a DBT usually translates binary code segments of a small granularity (e.g., a basic block or a trace). Based on the limited information from the code segment, a DBT may not always be able to determine whether the transformation would violate dependence at translation time. Moreover, a DBT usually has difficulty to recover such semantic information because the information could be lost in the binary level or the overhead of information recovery is unacceptable for a DBT. Thus, a conditional translation approach using runtime check for memory pointers is required if the dependence cannot be verified at translation time. For example, the pointer addresses of SIMD loads/stores in Fig. 1(c) are all unknown. If the addresses that are passed by parameter z and x have a dependence distance of 16 bytes, the translated AVX2 loop (vector size is 32 bytes) should not be executed and an additional check must be done to prevent it execution.

Third, the SIMD architecture usually requires that data be aligned to a certain boundary for efficient memory accesses, usually based on the vector length. As a result, the memory accesses with the short vector are restricted to a small value

```
void
foo (float *x, float *y, float *z)
{
  for (int i = 0; i < 1028; i++)
    z[i] += x[i] * y[i];
}
```

(a) Source code

```
L1:
    vld1.32  {d16-d17}, [r3]
    vld1.32  {d18-d19}, [r1]!
    vld1.32  {d20-d21}, [r2]!
    vmla.f32 q8, q10, q9
    vst1.32  {d16-d17}, [r3]!
    cmp      r3, r0
    bne      L1
```

(b) ARM32 NEON assembly

```
ShortSIMDLoop (SSL):
  %ptr1 = phi [ %r1, %Entry ], [ %ptr1.next, %SSL ]
  %ptr2 = phi [ %r2, %Entry ], [ %ptr2.next, %SSL ]
  %ptr3 = phi [ %r3, %Entry ], [ %ptr3.next, %SSL ]
  %vec1 = load <4 x float> * %ptr1      // x[i:i+4]
  %ptr1.next = add %ptr1, 16
  %vec2 = load <4 x float> * %ptr2      // y[i:i+4]
  %ptr2.next = add %ptr2, 16
  %vec3 = load <4 x float> * %ptr3      // z[i:i+4]
  %vec4 = fmul <4 x float> %vec1, %vec2
  %vec5 = fadd <4 x float> %vec3, %vec4
  store %vec5, <4 x float> * %ptr3
  %ptr3.next = add %ptr3, 16
  %cond = icmp eq %ptr3.next, %end.4112
  br %cond, label %Exit, label %SSL
```

(c) Short-SIMD loop IR

```
LongSIMDLoop (LSL):
  %ptr1 = phi [ %r1, %Entry ], [ %ptr1.next, %LSL ]
  %ptr2 = phi [ %r2, %Entry ], [ %ptr2.next, %LSL ]
  %ptr3 = phi [ %r3, %Entry ], [ %ptr3.next, %LSL ]
  %vec1 = load <8 x float> * %ptr1      // x[i:i+8]
  %ptr1.next = add %ptr1, 32
  %vec2 = load <8 x float> * %ptr2      // y[i:i+8]
  %ptr2.next = add %ptr2, 32
  %vec3 = load <8 x float> * %ptr3      // z[i:i+8]
  %vec4 = fmul <8 x float> %vec1, %vec2
  %vec5 = fadd <8 x float> %vec3, %vec4
  store %vec5, <8 x float> * %ptr3
  %ptr3.next = add %ptr3, 32
  %cond = icmp eq %ptr3.next, %end.4096
  br %cond, label %Exit, label %LSL
```

(d) Long-SIMD loop IR

Fig. 1: An example of NEON to AVX2 loop transformation. (Vector length is doubled)

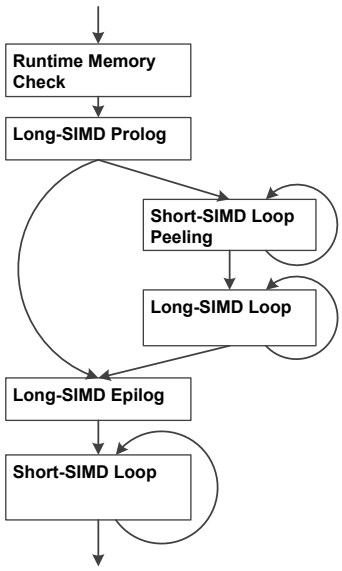


Fig. 2: Block organization.

of the alignment boundary but the long vector needs a larger value. Thus, memory accesses which are aligned in the original binary may become misaligned after transformation. For example, assume the situation where input array x is aligned at the 16-byte boundary but not at the 32-byte boundary. The accesses to x in the short-SIMD loop are always aligned, but those with a long-SIMD load (AVX2) are always misaligned because x is not aligned at the 32-byte boundary. Such a misalignment penalty could nullify the performance gain from the enhanced parallelization of the long-SIMD loop.

In the following subsections, we present the approaches for control management and legality checking. The details of the alignment problem are discussed in Section III.

B. Approach of the SIMD Loop Transformation

The input of the SIMD loop transformation is IR of a short-SIMD loop. We first scan the IR to see if SIMD memory patterns are contiguous. This work only considers contiguous memory accesses and we leave the optimization of interleaved data accesses for future work. After the IR is verified, the short-SIMD loop IR can be transformed to the long-SIMD loop, though it might not be executed due to violation of the memory dependence. In front of the short-SIMD loop body, we add five additional basic blocks: RuntimeMemoryCheck, LongSIMDProlog, ShortSIMDLoopPeeling, LongSIMDLoopBody and LongSIMDEpilog. Figure 2 shows the block diagram and control flow of the transformed blocks.

In the following description, we use the terms $ShortVS$ and $LongVS$ to respectively represent the vector size of the short and long vectors. $Expand$ represents the expanded vector factor which is computed from

$$Expand = LongVS \ / \ ShortVS.$$

Before generating the long-SIMD loop, we must compute the start and end iteration of the long-SIMD loop. From analyzing the indexing variable (i.e., a PHI node) of the short-SIMD loop, we can retrieve three values – $ShortStart$, $ShortEnd$ and $ShortStep$ which are respectively the start iteration, end iteration and step value of the short-SIMD loop. The three values $LongStart$, $LongEnd$ and $LongStep$ of the long-SIMD loop can be computed with

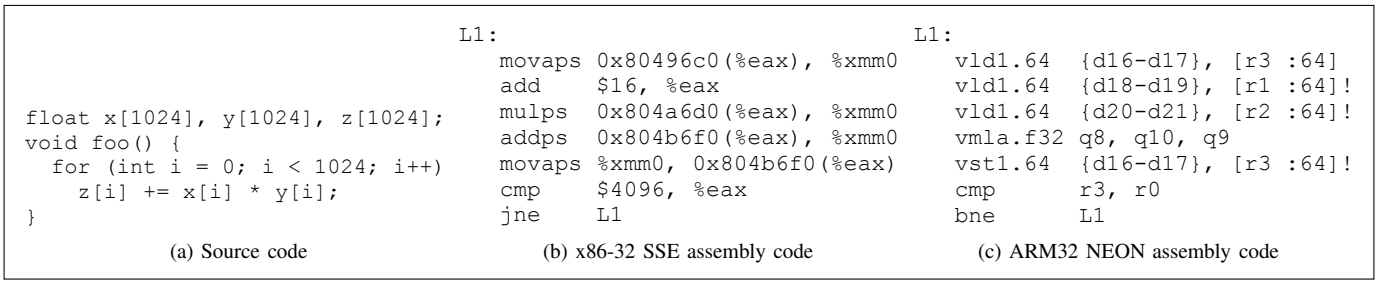
$$LongStart = ShortStart,$$
$$LongEnd = ShortEnd - ((ShortEnd - ShortStart)\%LongVS),$$
$$LongStep = LongVS,$$

and the calculations are inserted in LongSIMDProlog. Now, the long-SIMD loop body is populated by replicating the IR instructions of the short-SIMD loop with the following adjustments: the iteration of the long-SIMD loop is replaced with the computed three values, and the number of vector elements is multiplied by $Expand$ for each vector type.

Using the motivating example in Fig. 1(c), with transformation from NEON to AVX2 the value $Expand$ is 2 (i.e., 32-byte/16-byte). The short-SIMD loop's three values ($\%r3$, $\%end.4112$, 16) are found by scalar evolution analysis with the indexing variable, $\%ptr3$. The long-SIMD loop (Fig. 1(d)) is generated with the new three values, ($\%r3$, $\%end.4096$, 32), and the number of vector elements is updated to 8.

In the LongSIMDProlog block, in addition to computing the three values of the long-SIMD loop, we add a small test for the number of iterations, $LongEnd - LongStart$. We force the execution directly going to the short-SIMD loop instead of the long-SIMD loop if the iteration count is too small.

In the LongSIMDEpilog block, we need to calculate the resuming iteration from which the short-SIMD loop will start. This resuming value can be either $ShortStart$ if it comes directly from LongSIMDProlog or the remaining iteration, $LongEnd$, from the long-SIMD loop. The short-SIMD loop



(a) Source code (b) x86-32 SSE assembly code (c) ARM32 NEON assembly code

Fig. 3: A loop example of global arrays compiled with x86-32 SSE and ARM32 NEON.

will resume from this value or just skip if the long-SIMD loop has finished all loop iterations. In the example of Fig. 1(d), the long-SIMD loop can only run until the end value 4096 is reached, and the short-SIMD loop will finish the last iteration.

The aforementioned indexing variable is also an induction variable. For inductions other than the indexing variable, e.g., $\%ptr1$ and $\%ptr2$, the rewriting scheme is similar to that of the indexing variable. Vector reduction variables are handled similarly to traditional compilers for scalar to vector transformation. The rewriting scheme is as follows: (1) A long-SIMD variable is initialized in LongSIMDProlog for each long-SIMD reduction, and its initial value is set up by the initial value of the corresponding short-SIMD reduction. (2) The long-SIMD loop performs vector reduction operations on such long-SIMD variables. (3) In LongSIMDEpilog, the resultant long-SIMD reduction is split into partial short-SIMD reductions. The final result is computed by performing short-SIMD reduction operations on all partial reductions. (4) The resuming reduction value of the short-SIMD loop is updated with the final result computed in (3).

C. Runtime Memory Verification

Considering memory dependence, a runtime memory check of pointers is required if the DBT cannot determine from the short-SIMD loop IR whether the transformed long-SIMD loop can be executed or not. For such a case, checking codes are inserted in the RuntimeMemoryCheck block to conduct memory dependence verification. Given a group of SIMD store pointers and a group of SIMD load pointers collected from the short-SIMD loop, the distance of each load-and-store address pair is checked against the long-SIMD vector length, $LongVS$. Any violation to the dependence will force the execution to jump directly to the short-SIMD loop.

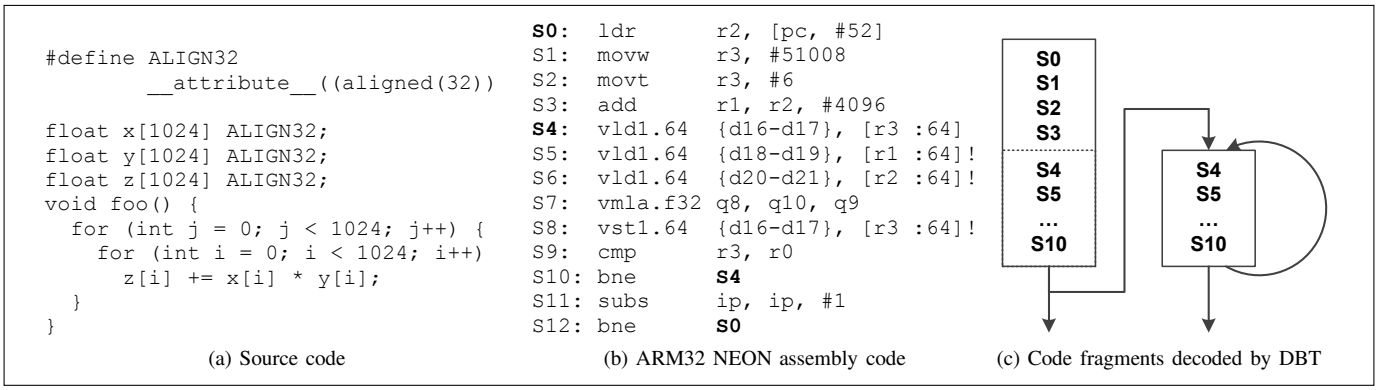
The runtime memory check can be optimized out if the DBT can compute the distances at translation time. For example, different from the source code in Fig. 1(a), the arrays in Fig. 3(a) are global arrays. In this case, the base addresses of such global arrays are directly encoded in the x86 instructions within the SSE loop, as shown in Fig. 3(b). By analyzing the short-SIMD loop IR instructions, the DBT can compute access patterns of the memory pointers, e.g., 0x80496c0+$\%eax$, 0x804a6e0+$\%eax$ and 0x804b700+$\%eax$, and conclude that the dependence distances are always of fixed large values and never violated. The short SSE loop can be safely transformed to AVX2 without inserting any runtime check code.

Figure 3(c) shows the ARM32 NEON binary compiled from the same loop source code. Different from SSE binary, array base addresses are not encoded in the ARM instructions because such address encoding is not supported by the addressing mode of NEON load/store instructions. Instead, the addresses are loaded into general-purpose registers before entering the loop body (e.g., $\%r1$, $\%r2$, and $\%r3$ in this case). As a result, address information of the arrays is missing in the short-SIMD loop IR. The DBT cannot verify the dependence distance at translation time, and thus runtime memory check is needed.

III. Alignment Problem

Recall that during the SIMD loop transformation, the short-SIMD load/store instructions are rewritten to long-SIMD load/store instructions by expanding the access size. However, the long-SIMD loads/stores can incur a significant performance penalty if the accessed data is not properly aligned with the long-SIMD ISA. This is because most SIMD architectures are designed with alignment constraint, usually based on the vector length, e.g., 16-byte boundary for NEON/SSE and 32-byte for AVX2. The accessed data of the short-SIMD binary are aligned according to the short-SIMD alignment constraint. When running on a long-SIMD machine, the data are very likely to be placed at memory locations with old alignment constraints and may not satisfy the alignment constraints of the long-SIMD ISA. Hence, data accesses with long-SIMD loads/stores become misaligned.

This misalignment situation can be relieved if the data of the source program are placed at a larger alignment boundary on the long-SIMD machines. As the data's alignment can satisfy the long-SIMD platform's alignment requirement, the transformed long-SIMD loads/stores will not cause any misaligned accesses. For example, the programmer can force data to align at 64-byte boundaries. The transformed binary can run on AVX2 and AVX512 machines (the alignment boundaries are 32- and 64-byte, respectively) without incurring misaligned accesses. To verify this scenario, we perform an experiment on a nested loop with NEON to AVX2 translation. The source code is listed in Fig. 4(a). In the source code, the global arrays are aligned at the 32-byte boundary which matches the alignment constraint of AVX2. The transformed long-SIMD loop should ideally have no misaligned memory accesses to the global arrays. Surprisingly, the long-SIMD loads/stores are all misaligned (profile data is listed in Section IV-C). The details of this problem are described in the next subsection.



(a) Source code (b) ARM32 NEON assembly code (c) Code fragments decoded by DBT

Fig. 4: A nest loop example with transformation from NEON to AVX2.

A. DBT Termination Condition

When an un-translated binary code is encountered, a DBT starts to decode and translate a fragment of binary code. In the decode phase, a DBT has to decide when the binary decoding should be terminated. A DBT usually stops decoding binary code when a control transfer instruction is encountered, e.g., a branch, call, or return instruction. Such a termination condition based on control transfer instructions is widely applied in many DBTs. Fig. 4(b) shows the ARM32 NEON assembly code of the nested loop in Fig. 4(a). As Fig. 4(b) shows, the inner loop is composed of the instructions from S4 to S10, and the outer loop is from S0 to S12. The DBT first decodes the outer loop starting from S0 until the first control transfer instruction, S10, is reached—thus the first code fragment, S0 to S10, is translated. Then, the instruction S10 branches to S4 (i.e., head of the inner loop) and the second code fragment, S4 to S10, is decoded because the code starting from S4 is not yet translated. The second code fragment forms a loop since S4 is a branch target of instruction S10, but the first code fragment is not because it jumps to the middle of the fragment. The two code fragments and their control flows are illustrated in Fig. 4(c). Since the second code fragment is a loop, it is further optimized with our SIMD loop transformation.

We can see from Fig. 4(c) that both code fragments consist of the binary code of the inner loop (S4 to S10). When the execution starts from the outer loop, one iteration of the inner loop is executed by the first code fragment. As a result, the second code fragment, which is transformed to a long-SIMD loop, will execute the inner loop starting from the second iteration. Hence, the actual addresses accessed by long-SIMD loads/stores are *shifted* by one short-SIMD vector length. The memory accesses become misaligned even though the global arrays are aligned at the long-SIMD alignment boundary.

Such a misalignment is caused by *code duplication* of the inner loop, and it can be overcome by preventing code duplication. The solution is to let a DBT stop binary decoding if it tries to decode an inner loop in the middle of a code fragment. For example, the decoding stops at point S3 with the example code. In the binary level, however, it is not easy for a DBT to determine whether the next instruction to decode is at a loop head or not. Even though the loop head information can be retrieved (e.g., with compiler annotation), this solution works well only when the data layout of the short-SIMD binary satisfies the alignment constraint of the long-SIMD ISAs. It cannot solve cases in which data are aligned at a smaller alignment boundary than the minimum constraint of the long-SIMD ISAs. Hence, a general approach is called for.

B. Dynamic Loop Peeling

A general solution to enforce the alignment of the long-SIMD loop is to dynamically peel the short-SIMD loop until the memory references are aligned with the long-SIMD alignment constraint before entering the long-SIMD loop. Hence, we create a block, ShortSIMDLoopPeeling, in front of the long-SIMD loop body. The ShortSIMDLoopPeeling block is populated by replicating IR instructions from the short-SIMD loop. The next step is to decide the number of iterations for short-SIMD loop peeling. The peeling count is calculated with a *voting* algorithm as follows:

All SIMD load/store pointers are collected from the short-SIMD loop. For each memory pointer, the distance between the referenced address and next long-SIMD alignment boundary is computed; the peeling count is the value of the distance divided by the short-SIMD vector length, as the formula

$PeelCount = (LongVS - P \ \% \ LongVS) \ / \ ShortVS$,

where P is the referenced address. Then peeling counts of all memory pointers are conducted by the voting process to find the majority, which is the number of short-SIMD loop iterations to be executed in ShortSIMDLoopPeeling. With this voting scheme computed in RuntimeMemoryCheck, we can ensure that as many long-SIMD loads/stores as possible are aligned. The algorithm is listed in Algorithm 1.

IV. Performance Evaluation

All experiments are performed on a system with one 6-core Intel Haswell Core i7-5930k 3.50 GHz processor. The host machine has 16 GB of main memory and the operating system is 64-bit Gentoo Linux with kernel 3.6.15. We conduct the SIMD loop transformation from the guest ISA, ARM32 NEON, to two long-SIMD host ISAs, x86-64 AVX2 and x86-64 AVX512. The LLVM compiler in our DBT only supports AVX512 of Knights Landing and no hardware is currently available to execute the Knights Landing binary, hence, we

Algorithm 1: Dynamic Loop Peeling

input : A list of basic blocks B in loop body, short-SIMD vector length w, long/short SIMD expanded factor s.
output: The number of iteration to peel k.

$K \leftarrow$ array of size s with zero initial values.
foreach *Basic block* $b \in B$ **do**
  foreach *Memory reference* $p \in b$ **do**
    $k \leftarrow (s\,w - (p \bmod (s\,w)))\,/w$
    $K[k] \leftarrow K[k] + 1$
  end
end
return $\arg\max_k K[k]$

run the AVX512 experiments using the Intel Software Development Emulator (SDE) version 7.31.0.

Twelve benchmarks are selected from two popular loop benchmark suites, Livermore loops (LL) [10] and the TSVC benchmark suite [11]. These two benchmark suites have often been used to evaluate vectorizing compilers. The SPEC benchmark suite is not selected in the experiment because its vectorization ratio is very low. All benchmarks are compiled using the ARM GCC 4.8.4 with flags "-marm -march=armv7-a -mtune=cortex-a8 -static -O3 -fno-unroll-loops -mfpu=neon -ffast-math -ftree-vectorize." Since ARM32 NEON does not support double-precision floating-point SIMD operations, the benchmarks are configured with single precision for floating-point data type. For comparison, we use the binary translation without SIMD loop transformation as our performance baseline. In the baseline translation, ARM32 NEON instructions are translated to x86-64 SSE instructions on the host machine.

A. Performance Results with AVX2

Figure 5 shows the speedup of NEON to AVX2 loop transformation. With FMA instruction generation disabled in LLVM JIT, the performance of NEON to AVX2 loop transformation achieves an average speedup of 1.41X compared with that without SIMD loop transformation. Many of the benchmarks exhibit an improvement in excess of 50%, while two benchmarks, `LL-inner` and `LL-PIC_1D`, show the least performance gain. The performance which can be achieved after loop transformation is related to two factors: (1) the percentage of time executed in the short-SIMD loop over total execution time, and (2) the composition of instructions in the short-SIMD loop. For instance, the benchmark `LL-inner` computes the inner product of two arrays. The offline compiler splits this computation into two loops: the first loop multiplies corresponding elements of the two input arrays and stores the results in an output array; the second loop iterates over each element of the output array and computes the final sum. Of the two loops, only the first loop is vectorized by the compiler and the second loop has no SIMD operations.

We profile the execution of `LL-inner` without applying SIMD loop transformation. The profile data reports the first loop contributes to only about 15% of the total execution time, which is the maximum performance gain our optimization can achieve. Moreover, the first loop is composed of only one vector multiplication and three vector loads/stores. For SIMD

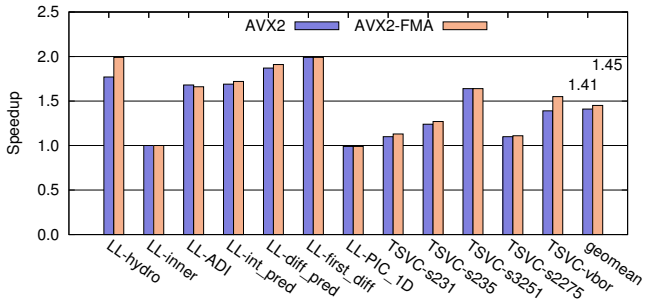


Fig. 5: Performance of NEON to AVX2 loop transformation.

memory accesses, since the array size of the benchmark is fixed, the number of outstanding memory access requests is the same regardless of whether the SIMD load/store instructions are translated with SSE or AVX2. For SIMD computation instructions, rewriting them improves the performance with longer SIMD lanes, however, the amount of SIMD computations in the loop is not sufficient. As a result, the performance is bounded by the SIMD loads/stores due to the high memory stall cycles. There is very little room for improvement on the first loop, thus no performance improvement is achieved after transformation. Similar behavior is also observed with benchmark `LL-PIC_1D`. In contrast, benchmarks such as `LL-first_diff`, `LL-hydro` and `TSVC-s3251` achieve significant performance improvements because their short-SIMD loops consist of a high ratio of SIMD computations.

To evaluate the impact of the host machine's multiply-and-add instruction, we compare the performance of enabling/disabling the FMA code emission in the LLVM JIT compiler. Note that there is no special FMA IR instruction supported in LLVM IR. Therefore, although the ARM32 NEON ISA supports the FMA feature, a multiply-and-add instruction in the ARM guest binary is translated into two LLVM IR instructions—a multiply instruction followed by an add instruction. The LLVM instruction selection pass can find such patterns and map them to one host FMA instruction.

Figure 5 also illustrates the transformation results from NEON to AVX2 with FMA code emission enabled. Performance improvement can be expected if the short-SIMD loop consists of multiply-and-add operations. `LL-hydro` and `TSVC-vbor` have the most significant performance gain with FMA because they are the benchmarks with the greatest number of multiply-and-add operations, and improve by about 13%. The overall benchmark performance is further improved to 1.45X on average with benefits from host FMA instructions.

The results in this subsection show the effectiveness of our DBT system: not only can we exploit the improved parallelism from long-SIMD architectures with our SIMD loop transformation, but we also allow the applications to use new SIMD features with the DBT technique.

B. Performance Results with AVX512

This experiment evaluates the loop transformation performance from NEON to AVX512, running with the Intel SDE functional emulator. Since the Intel SDE does not provide tim-

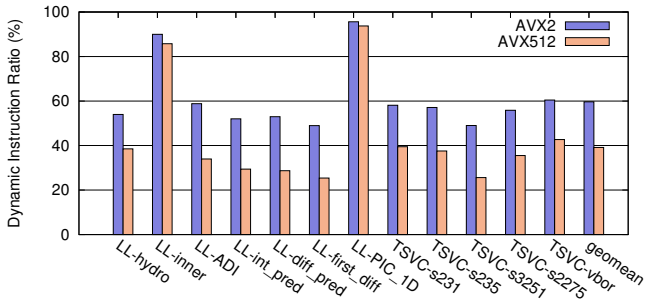


Fig. 6: Ratio of dynamic instructions with NEON to AVX2/AVX512 loop transformation.

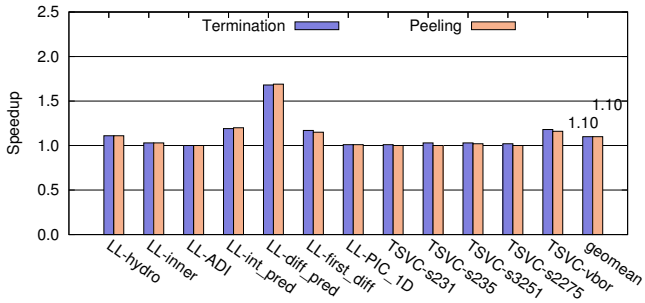


Fig. 7: Comparison of alignment optimization approaches.

ing information and no cycle-accurate simulator is currently available for AVX512, the number of dynamic instructions executed is measured here as the evaluation metric. Figure 6 shows the results normalized to the baseline performance (i.e., dynamic instruction count of NEON to SSE translation). Lower results are better. The results of the NEON to AVX2 transformation are also provided for comparison. The results show that fewer instructions are executed when a longer vector is used. On average, AVX2 and AVX512 respectively reduces the numbers of dynamic instructions executed to 60% and 39% of that without loop transformation. The dynamic instruction ratios of `LL-inner` and `LL-PIC_1D` are only slightly reduced. This is because the SIMD loop only contributes to a small proportion of the total instruction count. The dynamic instruction results of these two benchmarks also explain the results in Fig. 5 where no noticeable performance improvement is observed for either benchmark.

C. Comparison of Alignment Adjustment Approaches

We compare the performance of the SIMD loop transformation with three configurations: (1) no alignment adjustment, (2) optimized DBT termination condition, and (3) dynamic loop peeling. The experiment is conducted with NEON to AVX2 loop transformation. Because the alignment constraints of the guest and host SIMD ISA are different, we manually align source arrays at the addresses based on guest alignment constraint (i.e., NEON with 16-byte boundary but not 32-byte boundary) or on host alignment constraint (i.e., AVX2 with 32-byte boundary), and compare their performance. Since the alignment requirement is the 32-byte boundary for the host AVX2 ISA, we profile the benchmarks to see if the transformed AVX2 vector loads/stores are properly aligned at this boundary. We measure the misalignment ratio as

$$Ratio = \frac{\text{Number of misaligned SIMD accesses}}{\text{Number of total SIMD memory accesses}}.$$

Table I lists the misalignment ratio of the three configurations. When source arrays are aligned based on the host AVX2 constraint (i.e., the 32-byte boundary), the vector loads/stores in the long-SIMD loop ought to be aligned. However, the profile data show a 100% misalignment ratio for all benchmarks without applying any alignment adjustment approach. These surprising results are caused by the DBT code duplication problem and all accessed addresses become misaligned by one NEON vector length. By optimizing the DBT termination condition, most misaligned memory accesses are eliminated. The dynamic loop peeling approach also produces the same good results as the DBT termination condition optimization. Both approaches can achieve the least misalignment ratio. In Table I, there are still misaligned accesses, e.g., an 86% misalignment ratio with benchmark `LL-ADI`. The reason is that those SIMD load/store patterns originally misaligned in the guest binary remain misaligned after the SIMD loop transformation, e.g., a vector load that always accesses at 4-byte aligned addresses. They cannot be eliminated by our alignment adjustment approaches. Figure 7 shows the performance speedup of two alignment adjustment approaches against that without adjustment based on the AVX2 alignment constraint. Benchmark `LL-diff_pred` achieves a 70% performance improvement with alignment adjustment and an average 10% improvement is achieved for all benchmarks.

When source arrays are aligned based on the guest NEON constraint (16-byte boundary), one expects that the transformed long-SIMD loads/stores will be misaligned. However, the code duplication caused by DBT coincidentally causes misaligned accesses to become aligned. On the contrary, applying the optimized DBT termination condition reverses this situation. We can see from Table I that the misalignment ratio of the optimized DBT termination condition is higher than that without adjustment for several benchmarks. With dynamic loop peeling, the DBT always adjusts long-SIMD loops to start accessing arrays from aligned addresses. Therefore, the dynamic loop peeling approach can achieve the minimum misalignment ratio regardless of whether the source arrays are aligned with the guest or host SIMD alignment constraint.

Table II further demonstrates the benefit of dynamic loop peeling. In this experiment, the NEON to AVX512 loop transformation is conducted. We do not force the alignment of source arrays, but rather allow the guest compiler to determine the placement of source arrays. The misalignment ratios of the three configurations are listed in Table II. As the vector length of the host SIMD increases to 4 times that of the guest, the optimal peeling problem becomes more complex than a simple "peeling or not" decision as in the case of the NEON to AVX2 transformation. Therefore, changing the terminating condition is not as helpful since the available peeling distance becomes $k = \{0, 1, 2, 3\}$ in AVX512. It has only a 0.25 probability instead of 0.5 to "guess" the optimal one, while the dynamic

TABLE I: Misalignment ratio of (1) no adjustment, (2) optimized DBT termination condition and (3) loop peeling with NEON to AVX2 loop transformation.

	Aligned with NEON constraint (16-byte boundary)			Aligned with AVX2 constraint (32-byte boundary)		
Benchmark	NoAdjust	Term.	Peeling	NoAdjust	Term.	Peeling
LL-hydro	75%	75%	75%	100%	50%	50%
LL-inner	67%	33%	33%	100%	0%	0%
LL-ADI	86%	95%	86%	100%	86%	86%
LL-int_pred	0%	100%	0%	100%	0%	0%
LL-diff_pred	0%	100%	0%	100%	0%	0%
LL-first_diff	67%	67%	67%	100%	33%	33%
LL-PIC_1D	0%	100%	0%	100%	0%	0%
TSVC-s231	0%	100%	0%	100%	0%	0%
TSVC-s235	75%	25%	25%	100%	0%	0%
TSVC-s3251	43%	71%	43%	100%	14%	14%
TSVC-s2275	75%	25%	25%	100%	0%	0%
TSVC-vbor	43%	57%	43%	100%	0%	0%

TABLE II: Misalignment ratio with NEON to AVX512 loop transformation.

Benchmark	NoAdjust	Term.	Peeling
LL-hydro	75%	75%	50%
LL-inner	67%	33%	33%
LL-ADI	92%	92%	87%
LL-int_pred	100%	100%	0%
LL-diff_pred	100%	100%	0%
LL-first_diff	67%	67%	67%
LL-PIC_1D	100%	100%	0%
TSVC-s231	100%	100%	0%
TSVC-s235	100%	100%	75%
TSVC-s3251	86%	86%	57%
TSVC-s2275	100%	100%	75%
TSVC-vbor	86%	86%	57%

peeling algorithm always selects the optimal $\hat{k} \in k$. The profile data listed in Table II show that the dynamic loop peeling approach can achieve the minimal misalignment ratio.

V. Related Work

Pajuelo et al. [12] proposed a DBT system which conducts auto-vectorization from scalars to vectors. Their system requires new hardware support to speculatively execute vectorized code and fall back to scalar code when an invalid vectorization is detected by the hardware. In contrast, our DBT conducts translation from short-SIMD to long-SIMD, which is a pure software solution. A closely related work is [13] which also uses a dynamic binary rewriting technique to convert short-SIMD binary to long-SIMD code. Their work is implemented in a same-ISA DBT for the x86 architecture. If a short-SIMD memory operation could become misaligned in a long-SIMD form, their DBT system does not emit a long-SIMD memory instruction but use multiple short-SIMD memory accesses with packing. In contrast, our approach is designed in a retargetable DBT that supports cross-ISA translation. Our new contribution is that we propose using a general machine-independent IR layer for SIMD loop transformation and optimization. A general approach to dynamic loop peeling is proposed to enforce the alignment of long-SIMD memory accesses. Our work is more portable than a same-ISA DBT.

VI. Conclusion

In this paper, we present an approach of short-SIMD to long-SIMD loop transformation using a dynamic binary translation technique. We propose a general approach that translates the short-SIMD binary to machine-independent IR, conducts SIMD loop transformation at this IR level, and finally translates to long-SIMD instructions. We also identify the alignment problem where memory accesses aligned in the original binary may become misaligned when using long-SIMD instructions. The code duplication problem caused by DBT can further exacerbate the misalignment. The proposed approach of changing DBT decode termination conditions relieves such misalignment. A general solution using dynamic loop peeling is also presented, which can minimize misalignment. We demonstrate that the SIMD loop transformation can significantly improve the performance of short-SIMD binaries, not only from better parallelism of the long-SIMD architectures but from exploiting new SIMD hardware features. The proposed concept can also be applied to same-ISA DBTs, static binary translators and other JIT systems.

Acknowledgment

This work is supported in part by Ministry of Science and Technology of Taiwan under grant number MOST-104-2218-E-002-032 and MOST-102-2221-E-001-034-MY3.

References

[1] A. Peleg, S. Wilkie, and U. Weiser, "Intel mmx for multimedia pcs," *Communications of the ACM*, vol. 40, no. 1, pp. 24–38, 1997.

[2] A. J. C. Bik, *Software Vectorization Handbook, The: Applying Intel Multimedia Extensions for Maximum Performance*. Intel Press, 2004.

[3] M. Hassaballah, S. Omran, and Y. B. Mahdy, "A review of simd multimedia extensions and their usage in scientific and engineering applications," *The Computer Journal*, vol. 51, no. 6, pp. 630–649, 2008.

[4] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference*, 2005, pp. 41–46.

[5] J. Li, Q. Zhang, S. Xu, and B. Huang, "Optimizing dynamic binary translation for simd instructions," in *International Symposium on Code Generation and Optimization*, 2006, pp. 269–280.

[6] L. Michel, N. Fournel, and F. Petrot, "Speeding-up simd instructions dynamic binary translation in embedded processor simulation," in *Proceedings of the 2011 Design, Automation & Test in Europe Conference & Exhibition*, 2011, pp. 1530–1591.

[7] N. Clark, A. Hormati, S. Yehia, S. Mahlke, and K. Flautner, "Liquid simd: Abstracting simd hardware using lightweight dynamic mapping," in *IEEE International Symposium on High Performance Computer Architecture*, 2007, pp. 216–227.

[8] S.-Y. Fu, D.-Y. Hong, J.-J. Wu, P. Liu, and W.-C. Hsu, "Simd code translation in an enhanced hqemu," in *IEEE International Conference on Parallel and Distributed Systems*, 2015, pp. 507–514.

[9] D.-Y. Hong, C.-C. Hsu, P.-C. Yew, J.-J. Wu, W.-C. Hsu, Y.-C. Chung, P. Liu, and C.-M. Wang, "HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores," in *International Symposium on Code Generation and Optimization*, 2012, pp. 104–113.

[10] F. H. McMahon, "The livermore fortran kernels: A computer test of the numerical performance range," Tech. Rep., 1986.

[11] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua, "An evaluation of vectorizing compilers," in *International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 372–382.

[12] A. Pajuelo, A. Gonzalez, and M. Valero, "Speculative dynamic vectorization," in *International Symposium on Computer Architecture*, 2002, pp. 271–280.

[13] N. Hallou, E. Rohou, P. Clauss, and A. Ketterlin, "Dynamic re-vectorization of binary code," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2015, pp. 228–237.