SIMD Code Translation in an Enhanced HQEMU

Sheng-Yu Fu*  
d03922013@csie.ntu.edu.tw  
Ding-Yong Hong†  
dyhong@iis.sinica.edu.tw  
Jan-Jan Wu†  
wuj@iis.sinica.edu.tw  
Pangfeng Liu*  
pangfeng@csie.ntu.edu.tw  
Wei-Chung Hsu*  
hsuwc@csie.ntu.edu.tw

* Department of Computer Science and Information Engineering National Taiwan University, Taiwan  
† Institute of Information Science Academia Sinica, Taiwan

Abstract — HQEMU is a multi-threaded and re-targetable dynamic binary translator built on top of QEMU and LLVM. It combines the fast and reliable code translation in the TCG (Tiny Code Generator) of QEMU and the rich optimizations in LLVM to achieve high performance for both short running and long running applications. One weakness of HQEMU lies in the lack of efficient SIMD instruction translation. This work investigates on how to remedy that. Two approaches have been designed and tested. One simple approach is to modify the help function to emit LLVM vector IR, and a more complete approach is to add a newly introduced vector IR in the TCG phase. Although both approaches can exploit the SIMD instructions of the host machine, the second and more complete approach has superior runtime as well as compile time advantages.

I. INTRODUCTION

Dynamic Binary Translation (DBT) [1] has been commonly used in cross-ISA process virtual machines [2] to enable system or application migration from one ISA to another [3]. QEMU [4] is a fast system and user mode emulator based on DBT technology. For example, the well-known and widely used Android emulator is implemented with a QEMU emulating ARM executables in Android environment on x86 based PC platforms.

DBT is a fast emulation method. It dynamically translates guest executables (such as ARM) to native instructions on the host (such as x86), and cache the translated native code in memory to avoid re-translation. The translated code runs many times faster than the traditional interpretation approach. However, to further speed up the emulation by generating highly optimized code has been challenging since optimizations require longer translation time, which is part of the runtime, and may lead to unreliable code if optimizations are not thoroughly tested. Normally, in DBT, short running applications prefer fast code generation while long running applications desire more optimized code.

HQEMU [5] is a multi-threaded and re-targetable dynamic binary translator built on top of QEMU and LLVM [6]. It launches two DBT threads running concurrently, one for the original QEMU TCG, another for the LLVM based optimizer. For short running applications, LLVM based optimization would have no impact on the TCG generated code since by the time when LLVM finish optimizations of a code fragment, the execution of the fragment in TCG generated code might have finished. For long running applications, the original TCG generated code could be replaced by redirecting the execution to the optimized code generated from LLVM. This approach effectively combines the fast code translation in the TCG (Tiny Code Generator) of QEMU with the rich optimizations in LLVM to achieve high performance for both short running and long running applications. As reported in [5], HQEMU runs SPEC 2006 benchmarks 2.4 to 4 times faster than the original QEMU for emulating IA-32 executables on x64 machines.

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>Slow down</th>
<th>SIMD Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMv7</td>
<td>17.35</td>
<td>ARMv7 45.25%</td>
</tr>
<tr>
<td>IA32</td>
<td>3.77</td>
<td>IA32 75.50%</td>
</tr>
</tbody>
</table>

Table 1. Slowdown comparing to native execution  
Table 2. Ratio of SIMD instructions to total instruction

However, we have observed one weakness of HQEMU in that it lacks SIMD (Single Instruction Multiple Data) instruction translation capability. If the guest executable contains frequent SIMD instructions (such as SSE in IA32, AVX in x64, and Neon in ARM), the generated code has much room for improvement.

Consider Linpack, for example, when migrating IA32/ARMv7 executables to run on x86-64 machines using HQEMU, such a weakness is exposed. Table 2 shows the dynamic SIMD instruction ratio of ARMv7
and IA32 executables. The ratio for IA32 is 75% and 45% for ARMv7 binary. The ratio for ARMv7 is lower since Neon instructions lack double precision versions. (In our opinion, ARMv8 has a more competitive SIMD instruction set). Table 1 shows the relative performance of ISA migration of HQEMU comparing to native execution. (ARMv7 native execution is running on a SAMSUNG EXYNOS5 ARMv7 board) For IA32 guest executables, HQEMU runs 3.77x slower than the native execution. Usually, on floating applications, HQEMU is only 2.1x slower than native execution. For ARMv7 guest executables, HQEMU runs 17x slower than native execution. This discrepancy triggers us to look into how HQEMU can be enhanced with efficient SIMD instruction translation.

SIMD instructions can compute multiple data in a single cycle. The execution of a SIMD instruction avoids the overhead of redundant instruction fetch, decode, data dependency checking, and writing results back to registers, so it is both power and execution efficient. SIMD instructions have been widely supported in modern microprocessors, such as SSE/AVX for X86, NEON for ARM, Altivec for PowerPC and MSA for MIPS. At the same time, SIMD code generation in compilers has also made significant progress such as SLP (SuperWord Level Parallelism) [7]. Polyhedral-Model Auto-Vectorization [8] and Whole-function vectorization [9], just to name a few. We expect to see a greater ratio of SIMD instructions to show up in current and future application executables. Hence we also believe it would be critical to enhance current DBT translators with efficient SIMD code translation.

In this work, we enhance HQEMU by designing and evaluating two SIMD instruction translation approaches. One is to modify the SIMD related helper functions to emit LLVM vector IR, and rely on the LLVM backend to generate SIMD instructions on the host machine. The second approach is to add vector IR to TCG so that the guest SIMD instructions could be converted into vector IR from early on. Having TCG to generate vector IR, the information of the guest SIMD could be better preserved and passed to backend. The second approach, although somewhat more complicated than the first, yield better results in terms of both compile time and runtime performance. Having vector IR early on would benefit translation time by compiling a shorter list of IRs (notice that one vector IR could represent multiple scalar IRs). However, adding such portable vector IRs in TCG is not trivial because it must deal with various guest SIMD architectures [10]. Furthermore, an ill-designed IR may impact the translation and optimization time, which are part of runtime, in DBT.

In this work, we have designed and implemented our vector IRs and other relevant optimizations in HQEMU. It can support multiple guest SIMD architectures including x86 SSE/AVX and ARMv7 NEON. The enhanced HQEMU runs SPEC CFP2006 and the Linpack benchmark much faster.

To summarize, this work makes the following contributions.
1.) We propose and implement two approach of SIMD translations to enhance the current HQEMU. One implementation involves only the LLVM backend, and the another approach requires adding a new set of vector IRs in TCG. The newly added vector IR currently supports x86 SSE/AVX and ARMv7 Neon.
2.) We have implemented and evaluated two different approaches and compare the strength and weakness of each approach. Adding vector IR in TCG yields better runtime as well as compile time performance.
3.) The enhanced HQEMU now runs a set of benchmarks with SIMD instructions 1.31x faster for emulating IA32 executables on x64, and 2.58x faster for emulating ARMv7 executables on x64. We believe the proposed SIMD translation approach could be applied to other DBT tools.

The rest of this paper is organized as follows: section 2 gives the background of this work. Section 3 is the design and implementation of our vector IR and describe how vector IR cooperates with HQEMU. Section 4 evaluates the effectiveness after applying our approaches and we also compares the performance with an approach that leveraging original TCG helper function. Section 5 discusses related work and Section 6 summaries and concludes.

Figure 1: The architecture of the HQEMU framework.

**II. BACKGROUND**

This work is based on HQEMU, which is built on top of QEMU and LLVM. Figure 1 illustrates the architecture of HQEMU. The two translators, running
on different threads, are designed for different purposes. The QEMU TCG focuses on simple and fast code translation to minimize the translation overhead. It conducts few simple optimizations (i.e., liveness analysis and store forwarding optimization) so the execution can be quickly switched to the translated code in the block code cache. When the emulation manager identifies that some code region is hot and worthy of further optimization, the LLVM optimizer is activated to dynamically improve the code region for higher performance. The rich set of analysis facilities and powerful optimizations in LLVM can generate very high-quality code in the trace code cache. This framework effectively combines the advantages of both translators and achieves high performance for both short running and long running applications. In addition, with the multi-threaded design, the high optimization overhead of the LLVM optimizer can be hidden without interfering with the guest application execution.

Two-level IR conversion The goal of HQEMU is to have a single DBT framework to take on application binaries from several different ISAs and re-target them to host machines with different ISAs. Using a common intermediate representation (IR) is an effective approach to achieve retargetability, which is used in both QEMU TCG and LLVM. By combining these two frameworks, HQEMU inherits their retargetability with minimum effort. In HQEMU, when LLVM optimizer receives an optimization request, the binary of guest ISA is not directly translated to LLVM IR. Instead, the guest binary is first converted to TCG IR, and then to LLVM IR. Such two-level IR conversion simplifies the translator tremendously because TCG IR only consists of about 125 different operation codes – much smaller than in most existing ISAs. Without such two-level IR conversion, for example, supporting full x86 ISA requires implementing more than 2000 x86 opcode to LLVM IR conversion routines.

Helper function inlining Helper functions are usually implemented with high-level languages (e.g., C or C++), which makes complex ISA semantics easier to implement. For example, QEMU TCG uses helper functions to emulate SIMD operations, x87 floating point operations and computation of x86 EFLAGS. However, invoking a helper function needs to pay function call overhead. The emulated guest states also need to be saved to memory before calling a helper function because we cannot be sure if the helper function would read or modify any guest states. HQEMU eliminates such overhead by inlining helper functions. Moreover, the LLVM IR after inlining can expose more opportunity for register mapping. In HQEMU, some profitable helper functions are manually selected and pre-compiled to LLVM IR with an offline compiler (i.e., Clang). When the TCG call opcode is translated by the LLVM optimizer, LLVM IR of the helper function is merged if the inlining is beneficial. For instance, the helper to compute x86 EFLAGS is designed with a big switch of 50 conditions. This helper is usually inlined because most conditions can be eliminated and thus only a short sequence of LLVM IR will be actually inlined.

In addition to traditional compiler optimizations, HQEMU also includes some DBT-related optimizations such as block chaining [11] and indirect branch translation caching [12] to avoid frequent switching between the DBT dispatcher and the code cache. Trace formation is also applied in HQEMU to increase the granularity of the LLVM optimization.

III. SIMD CODE TRANSLATION

SIMD instructions have been widely supported in modern microprocessors. Significant advances in SIMD code generation in compilers have also been reported. However, SIMD code translation has not attracted much attention in DBT, such as QEMU. The current QEMU does not support vector types in its TCG IR and it emulates guest SIMD instructions through helper functions. Furthermore, the helper function uses a sequence of scalar instructions to emulate a guest SIMD instruction instead of leveraging the SIMD instruction already supported by the host machine.

Figure 2 shows a SIMD translation example with QEMU TCG. The source code shown in Figure 2(a) performs a vector addition of four single-precision floating-point values. The instructions, addps and vadd.f32, are generated for x86 SSE and ARM NEON, and their translated code are shown in Figure 2(b) and 2(c), respectively. For the x86 guest, instruction addps is translated to one helper function call in the code cache. The helper function sequentially iterates over each vector element and conducts scalar floating-point addition for each computation. Different to the design of the x86 guest, the helper of the ARM guest is implemented with only one scalar floating-point addition. Hence, four helper function calls are emitted in the code cache to complete the whole vector operation.

The way QEMU emulates guest SIMD instructions also influences the translation capability on SIMD instructions in HQEMU. The reasons are as follows: (1) The LLVM optimizer depends on QEMU TCG frontend to translate guest binary code to LLVM IR with two-level IR conversion. Since QEMU TCG translates SIMD instructions to function calls to external helper functions, the LLVM optimizer also emits LLVM call instructions to invoke helper functions. For the example in Figure 2,
similar code sequence is emitted by the LLVM optimizer as QEMU does. (2) Recall that HQEMU can pre-compile SIMD related helper functions to LLVM IR. The generated LLVM IR of the SIMD helpers will consist of LLVM scalar IR only, because TCG implements them with the sequence of scalar instructions. Although the LLVM optimizer can inline SIMD helper functions, the emitted LLVM IR is still kept in its scalar form. As a result, no host SIMD instructions will be generated in the optimized code even if the LLVM IR has already supported vector types and operations.

To enhance the translation capability of HQEMU, we propose to generate LLVM vector IR so as to exploit SIMD instructions of the host machine. In the following sections, we elaborate on the design details of two different approaches that achieve efficient guest to host SIMD translations.

A. Rewriting SIMD Related Helper Functions

The reason that impedes the LLVM optimizer from generating host SIMD instructions is because the pre-compiled SIMD helpers consist of LLVM scalar IR only. Since LLVM has supported vector IR, one approach to overcome the problem is to modify SIMD related helper functions in order to emit LLVM vector IR.

We implement a SIMD rewriter in the LLVM optimizer. When the pre-compiled LLVM IR is fetched into HQEMU, the SIMD rewriter rewrites common SIMD helper functions, replacing scalar IR with LLVM vector IR. In the x86 addps helper example in Figure 2(b), the original four LLVM floating-point addition instructions are rewritten as one LLVM vector instruction, %result = add < 4 x float > %d, %es, where LLVM vector type < 4 x float > is used. Such SIMD helper function rewriting has the following advantages:

This simple solution can achieve minimal changes to HQEMU. Extension is only made to the LLVM optimizer, and no change is required for neither QEMU TCG frontend nor backend. Moreover, the guest SIMD helpers are re-written only once right after the application emulation begins. They can be reused for the rest of emulation time. The optimized LLVM vector IR is always beneficial to be inlined. The DBT performance is improved because with the SIMD-rewrite enhancement, the LLVM optimizer can emit host SIMD instructions in the code cache, which in turn also eliminates helper function call overhead.

B. Vector Support in TCG IR

SIMD helper function rewriting is a simple and effective approach to exploit vector facilities on the host machines; however, it has several limitations. First, the LLVM vector IR that can be generated by the SIMD rewriter is highly dependent on how the SIMD helper functions are implemented by QEMU TCG. Take the ARM guest translation in Figure 2(c) as an example. The emulation of ARM vadd.f32 instruction is split into four helper function calls, one addition for each call. For those helpers that only conduct parts of a SIMD operation, the SIMD rewriter is unable to replace them with LLVM vector IR. As a result, no LLVM vector IR is emitted for the ARM example in Figure 2(c). Second, the guest SIMD instructions are not all translated through helper functions. For example, the vector load/store instruction which loads/stores 128-bit data from memory is directly decomposed into four scalar load/store instructions loading /storing 32-bit data at one time. SIMD rewriter cannot be applied to SIMD instructions that are not translated through helper functions. Third, such approach can only enhance the
translated code quality from the LLVM optimizer. The code quality generated from QEMU TCG remains unchanged. Therefore, short running applications cannot benefit much from this approach.

To remove the limitations, we propose the second approach which adds vector IRs in TCG instead of using helper functions. Guest SIMD instructions are represented directly with TCG vector instructions, e.g., the example in Figure 2 will be translated to the same TCG vector instruction (i.e. vadd.f32.128) for both guest ISAs. Moreover, the TCG vector instructions can be explicitly converted to LLVM vector IR with the two-level IR conversion approach. Now that both the LLVM optimizer and QEMU TCG can emit vector IRs and thus fully enjoy the benefits from host SIMD architecture. Having TCG to generate vector IR in the early stage, it yields better performance not only for long running applications but also for short running ones. Adding portable vector IRs in TCG is not trivial because it must deal with various guest SIMD implementations. Furthermore, an ill-designed IR may impact the translation and optimization time in DBT. The considerations to add such portable vector IR are discussed in this section.

- **Vector IR in memory representation**

In general, intermediate representation is used to pass the information between two different layers. Thus, the in-memory representation may vary depending on the usage. For example, GCC GENIRIC [13] uses tree representation. There are also graph-based Intermediate Representation [14] and array-based, link-list based representation, etc. Compared to array and link list base IR, tree and graph base IR can store information more easily, and can easily adapt to larger program scope. However, tree or graph traversal is time consuming. Here, we will discuss which kind of IR is suitable for two-level dynamic binary translator. In this work, we consider the following design issues:

**Vector IR in two-level IR DBT:** Our vector IR is the extension of TCG IR (It also the extension of the first level IR of HQEMU). The first level IR should facilitate fast translation and performing light weight optimizations. In this case, array and link list is much more suitable.

**The scope in dynamic binary translation:** Since the input to a DBT is guest binary, the optimization scope in a DBT is limited to a basic block. The granularity can be increased by forming traces. Even with traces, the optimization scope seen by a dynamic binary translator is much smaller than the scope (e.g., loops, functions, procedures) visible to a compiler or static binary translator. Therefore, array or link list based representation is sufficient for a DBT.

Therefore, we maintain our vector IR in array data structure to achieve fast translation. Moreover, because our vector IR is compatible to the original TCG IR, optimizations applied on TCG IR, such as register liveness analysis and store forwarding optimization( Dead code elimination is done by performing these two optimization passes.) can also be applied on our vector IR.

- **Compound Vector IR Support**

A compound instruction can perform several operations within one instruction. For example, the x86 instruction pmaddwd (multiply and add packed integers) can perform an addition followed by a multiplication in one instruction. There are also studies about how to design high performance and power efficiency processor based on compound instruction [15]. In our vector IR design, the question is: should we support compound instructions in the vector IR? For example, the instruction pmaddwd is translated to one multiply-add IR instruction or is converted to two IR instructions, one multiplication plus one addition instruction? If supporting compound vector IR, the variety of different guest SIMD instructions would increase the amount of vector IR to be designed in TCG. However, in our two-level IR DBT, once the information is lost in the first level IR, it is difficult and time consuming to recover it in the later level. Although the optimizer could try to recognize the patterns which is composed of multiplication followed by an addition instruction, such pattern matching is difficult and time consuming, and is not suitable for a DBT system. Thus, Compound vector IR supporting is more helpful to DBT systems, especially for HQEMU, because the translation overhead is a critical issue on designing a high performance DBT system.

- **Opcodes Supported in TCG Vector IR**

In our current design, we implement about 60 different vector IR instructions by analyzing the two most popular ISAs, ARM and x86, and profiling a broad class of applications, including scientific, multimedia, and other popular benchmarks. The set of vector IR instructions we have implemented cover data processing (i.e. arithmetic and logical computation instructions), data movement and vector load/store instructions. Table 3 lists the type of vector IR supported.

In table 3, the vector instruction format is of OPCODE_<ElementType>_<VectorLength>. For example, instruction vorn_I28 is to conduct bitwise OR-NOT with 128-bit vectors (the element type is omitted with logical operations); vadd_i8_I28 is to perform sixteen 8-bit integer additions at the same time,
and instruction `vma_f32_128` performs multiply-add-on four single precision floating-point values.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instruction example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic operation</td>
<td>vand_128, vand_64, vorn_128</td>
</tr>
<tr>
<td>Integer arithmetic</td>
<td>vadd_i8_128, vadd_i64_128, vadd_i32_128, vadd_i64_128</td>
</tr>
<tr>
<td>Floating arithmetic</td>
<td>vma_f32_128</td>
</tr>
<tr>
<td>Bit mask</td>
<td>vbst_128</td>
</tr>
<tr>
<td>Load, Store and data movement</td>
<td>vload_128, vstore_128, vmov_128</td>
</tr>
</tbody>
</table>

Table 3. Opcodes Supported in TCG Vector IR

### IV. EXPERIMENTAL RESULTS

In this section, we evaluate the effectiveness of the proposed TCG vector IR. First, we describe the experimental environment and benchmark suite we use. Then, we compare the performance improvement of the translated code with vector IR support vs. helper function rewriting. We also discuss the impact of SIMD/scalar instruction ratios on the effectiveness of our optimization.

#### A. Experiment environment

We implement our vector IR in both IA32 and ARMv7 frontend. These two frontend have the same backend, x86-64. Thus our HQEMU is installed on x86-64 machine. Table 4 is the host machine, OS and HQEMU version. For benchmark, we select the popular SPEC 2006 CFP benchmark and Linpack. The data size of SPEC 2006 CFP is train input.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Intel(R) i7-5930K CPU @ 3.50GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Linux 3.16.5-gentoo</td>
</tr>
<tr>
<td>Benchmark set</td>
<td>SPEC 2006 CFP, Linpack</td>
</tr>
<tr>
<td>HQEMU version</td>
<td>Based on official QEMU 2.2.0</td>
</tr>
</tbody>
</table>

Table 4. Host machine environment and benchmark

The compilers we use for IA32 applications are gcc-4.8, gfortran-4.8 and g++-4.8; the compilers for ARM are arm-linux-gnueabihf-gcc-4.8, arm-linux-gnueabihfgfortran-4.8 and arm-linux-gnueabihfg++-4.8. The compilation flag is “-ftree-vectorize, -ftree-slp-vectorize” together with other vectorization flags. However, because ARM-v7 NEON doesn’t support double precision floating point operations, many double precision benchmarks generate less NEON instructions. In addition, if a statement contains double-precision-floating type operands, the whole statement will be translated to scalar instructions. For example, in Figure 3, even though only $d[i]$ is double precision floating point, gcc-4.8 still translates statement S1 to all scalar instructions.

$d[i]$ is double precision floating point, gcc-4.8 still translates statement S1 to all scalar instructions. Thus, many benchmarks containing double precision floating point operations generate no NEON instructions. As a result, the performance improvement is not noticeable.

For fair comparison, we replaced the double precision data type declared in some SPEC 2006 CFP with single precision. Such benchmarks are marked with the suffix `_sfp`. For example, GemesFDTD_sfp is the single precision version of GemesFDTD. The results of some benchmarks are not included in this paper because of compilation failures or runtime errors in the official QEMU.

#### B. performance improvement

Figure 4 and Figure 5 show the performance improvement by our optimization for SIMD instructions. We implement two approaches in original HQEMU to generate host SIMD instructions. The two approaches are helper function rewriting and adding vector IR. In both figures, the y-axis is the speed up comparing to original HQEMU. For every benchmark, the blue bar represents speed up with helper function rewriting approach; the red bar is the speed up by the vector IR approach. For example, for IA32 guest, as shown in Figure 4, leslie3d’s red bar means vector IR approach can run 1.37x faster than original HQEMU, vector copy can reach 2.03x and others benchmark can also reach 1.01x to 1.71x speed up. In Figure 5, only the bwaves, 1.05x, wrf, 1.52x, vector_copy, 2.04x, and other single precision floating (with _sfp suffix) benchmarks achieved good speed up. The reason is the lack of SIMD instructions in the guest binary. We also show the dynamic instructions distribution in Figure 6 and Figure 7. These SIMD instructions are categorized into data processing (blue area) and load/store/move (red area) instruction. The number marked in the green area is the proportion of other scalar instructions. Some benchmarks’ SIMD ratios.

![Figure 3. S1’s binary generated by gcc-4.8. Even though only d[i] is double precision floating point, gcc-4.8 still translates statement S1 to all scalar instructions.](image-url)
Figure 4. Performance improvement with IA32 guest

Figure 5. Performance improvement with ARMv7 guest

Figure 6. IA32 dynamic instruction distribution

Figure 7. ARMv7 dynamic instruction distribution
are pretty low. Without SIMD instructions, our enhancement has little impact. Therefore, we did not include those benchmarks with low SIMD ratios in Figure 4 and Figure 5.

C. In-depth investigation on some benchmarks

Linpack The benchmark can achieve 1.71x speed up when the guest is IA32. The main reason is the high percentage of SIMD instructions. The SIMD ratio to all executed instructions of IA32 guest is 75%. However, the benchmark achieves 26.4x speed up when the guest is ARMv7. The result is surprising because the benefit from SIMD instruction is bounded by 4(SIMD length is just 4). We use perf to profile the breakdown of execution time when HQEMU emulates the ARMv7 Linpack, as shown in Table 5.

<table>
<thead>
<tr>
<th>Helper function rewriting</th>
<th>Vector IR</th>
</tr>
</thead>
<tbody>
<tr>
<td>float32_muladd</td>
<td>56.87%</td>
</tr>
<tr>
<td>Code cache</td>
<td>14.98%</td>
</tr>
<tr>
<td>roundAndPackFloat32</td>
<td>11.02%</td>
</tr>
<tr>
<td>float32_squash_inps</td>
<td>11.01%</td>
</tr>
<tr>
<td>helper_vfp_muladd</td>
<td>4.29%</td>
</tr>
<tr>
<td></td>
<td>Code cache</td>
</tr>
<tr>
<td></td>
<td>50.43%</td>
</tr>
<tr>
<td></td>
<td>float32_muladd</td>
</tr>
<tr>
<td></td>
<td>9.28%</td>
</tr>
<tr>
<td></td>
<td>roundAndPackFloat32</td>
</tr>
<tr>
<td></td>
<td>3.58%</td>
</tr>
<tr>
<td></td>
<td>float32_squash_inps</td>
</tr>
<tr>
<td></td>
<td>3.53%</td>
</tr>
<tr>
<td></td>
<td>cpster_write</td>
</tr>
<tr>
<td></td>
<td>3.08%</td>
</tr>
</tbody>
</table>

Table 5. Time breakdown when emulating the ARMv7 Linpack

The vector IR approach (right column) spends 50.43% of total execution time in the code cache and the helper function rewriting approach spends 56.87% of total execution time in float32_muladd function. In addition, the helper function, helper_vfp_muladd (4.29%) is a wrapper. It calls float32_muladd. In other word, in this approach, HQEMU still spends more than 60% execution time in multiply-add helper functions although we had tried to inline helper functions. The reason is:

a. The emulation of ARM multiply-add SIMD instruction is split into four helper function calls, one multiplication and addition for each call. For those helpers that only conduct parts of a SIMD operation, the SIMD rewriter is unable to replace those with LLVM vector IR. As a result, no LLVM vector IR is emitted for the ARM multiply-add SIMD instruction. Without LLVM vector IR, the backend still generates software helper function call. Therefore, the main speedup combine the hardware floating point instructions and SIMD instructions.

b. As mentioned above, the helper_vfp_muladd is a wrapper. It is not a leaf function and cannot be inlined. These un-inlined helper functions impede optimizations.

In our experiment, if we turn off the State mapping optimization, the performance contrarily increases by 10%. Finally, after applying vector IR approach, HQEMU can run Linpack 1.52x faster than native execution on ARM board when the guest is ARMv7. When the guest is IA32, the emulation is only 2.20x slower than native execution on the IA32 machine. If comparing ARMv7 emulation with native execution on the IA32 machine, the ARMv7 emulation run 3.36x slower than the native execution.

Vector_copy This benchmark copy the source array to the destination array and thus most of SIMD instructions are load/store/move instructions. Both QEMU and HQEMU emulate SIMD load/store instructions with a sequence of scalar load/store instructions. Without helper function, the helper function rewriting approach doesn’t work. In this case, the vector load/store IR is necessary. Moreover, the vector load/store IR can also pass the alignment information to both TCG backend and LLVM optimizer such that the code generator can generate aligned load/store instruction easily.

Calculix Vector IR supporting approach can run 1.29x faster than helper function rewriting approach when the guest is IA32. Although the data size is train, the execution time of calculix is still short and basic block iteration count is not enough to generate trace. Without trace, the LLVM optimizer will not be triggered and thus HQEMU only uses TCG translator to translate guest instructions. The TCG translator of helper function rewriting approach does not leverage host SIMD instruction, so the speedup is significant. Moreover, in our observation, calculix’s proportion of optimization and translation to total execution time is much higher than other benchmarks. Thus, even if the LLVM optimizer is triggered, the emulation may be finished before generating the optimized code.

D. Optimization and compilation time comparison

In general, the optimization and compile time is a function of the total number of IRs. The larger the number of IRs, the longer the compilation time. The trace built by helper function rewriting is usually longer than built by vector IR. In addition, the code quality is also better than helper function rewriting. Good quality code can reduce the time spending in rescheduling, inserting and removing IR. In our experimental result, the optimization and compilation time of vector IR approach is usually less than helper function rewriting approach, as shown in Figure 8 and Figure 9.

E. ARMv8 instruction distribution


Because the ARMv7 doesn’t support double-precision floating operation, the NEON ratio in SPEC 2006 CFP benchmark is much lower than SSE ratio. ARM Company published ARMv8 architecture in previous year and had extend the NEON instruction set to support double precision. Thus, we also profile the ARMv8 instruction distribution. The benchmark is also SPEC 2006 CFP. Compiler are aarch64-linux-gnu-gcc-4.8, aarch64-linux-gnu-g++-4.8 and aarch64-linux-gnu-gfortran-4.8. The result is shown in Figure 10.

**V. RELATED WORK**

SIMD instructions are widely supported in modern processors and increasingly more popular in microprocessors. For example, Intel has increased the length of SSE registers from 128 bits to 256 bits in AVX, and further to 512 bits in x86 AVX512. ARMv8 also extends NEON instruction set to support the double precision. However, such ubiquitous SIMD instructions also have marked discrepancy between different ISAs. Thus, there are some works concentrate on how to generate SIMD instruction for various hardware [10]. Dorit Nuzman et al. propose Vapor SIMD [16]. Vapor SIMD compiles source files (.c files) to portable vectorized bytecode first. Then, the JIT compiles the bytecode to native code and executes it at runtime. Both [10][16] compile source files to vector bytecode or SIMD instructions in static time. For TCG vector IR extension, [17] is the related work. Their work extends the vector IR to QEMU TCG IR such that the DBT can generate SIMD instructions. For optimization of SIMD code in DBT, Li et al. propose a SIMD data type tracking algorithm to trace the type of data element in SIMD registers [18]. They focus on SIMD register mapping rather than general SIMD code translation. Their work can decrease...
the number of data movement and was implemented in IA-32 EL. However, for multi-level DBT, there has been little research on SIMD translation and optimization.

VI. CONCLUSIONS AND FUTURE WORK

HQEMU is a multi-threaded and re-targetable dynamic binary translator built on top of QEMU and LLVM. HQEMU runs SPEC 2006 benchmarks 2.4 to 4 times faster than the original QEMU for emulating IA-32 executables on x64 machines. However, one weakness of HQEMU lacks SIMD instruction translation capability. To enhance HQEMU with efficient SIMD translation, we have proposed two approaches: One is rewriting SIMD related helper functions in HQEMU to emit SIMD instruction on the host machine. It is a simple solution that requires minimal changes to enable SIMD instruction generation. Another approach is adding vector IR to the original QEMU TCG. This is a more complete approach and yields superior runtime and compile time performance. It can benefit both short and long running applications. The enhanced HQEMU with the second approach is now running Linpack and spec2006 CFP 1.31x faster for emulating IA32 executables on x64, and 2.58x faster for emulating ARMv7 executables on x64.

For the ARMv7 guest executables, some of the benchmarks contain few SIMD instructions because the NEON does not support double precision operations. We have also profiled the ARMv8 executables and observed that the SIMD instruction ratio is much higher than in the ARMv7 executable. We believe the enhanced HQEMU would perform even better on ARMv8 executables.

REFERENCE


[15] Smith, J. E. "FUTURE SUPERSCALAR PROCESSORS BASED ON INSTRUCTION COMPOUNDING."

