Improving SIMD Code Generation in QEMU

Sheng-Yu Fu
Department of Computer Science
and Information Engineering
National Taiwan University, Taiwan
d03922013 @csie.ntu.edu.tw

Jan-Jan Wu
Institute of Information Science
Academia Sinica, Taiwan
wuj@iis.sinica.edu.tw

Wei-Chung Hsu
Department of Computer Science
and Information Engineering
National Taiwan University, Taiwan
hsuwc@csie.ntu.edu.tw

***Abstract*— Modern processors are often enhanced using SIMD instructions, such as the MMX, SSE, and AVX instructions set in the x86 architecture, or the NEON instruction set in the ARM architecture. Using these SIMD instructions could significantly increase application performance, hence in application binaries a significant proportion of instructions are likely to be SIMD instructions. However, Dynamic Binary Translation (DBT) has largely overlooked SIMD instruction translation. For example, in the popular QEMU system emulator, guest SIMD instructions are often emulated with a sequence of scalar instructions even when the host machines have SIMD instructions to support such parallel computation, leaving significant potential for performance enhancement. In this paper, we propose two approaches, one leveraging the existing helper function implementation in QEMU, and the other using a newly introduced vector IR (Intermediate Representation) to enhance the performance of SIMD instruction translation in DBT of QEMU. Both approaches were implemented in the QEMU to support ARM and IA32 frontend and x86-64 backend. Preliminary experiments show that adding vector IR can significantly enhance the performance of guest applications containing SIMD instructions for both ARM and IA32 architectures when running with QEMU on the x86-64 platform.**

I. Introduction

QEMU [1] is a commonly used cross-ISA (Instruction Set Architecture) emulator. It can emulate cross-ISA application migration either on a per process basis (process virtual machine) or as a complete OS (system virtual machine). Several virtual iOS and Android smart phone environments running on x86 based PCs have recently been deployed on such cross-ISA system emulators. Also, for embedded application development, emulating the Android/ARM environment on the more powerful X86 PC platform allows for more convenient and productive application development and debugging. However, since the ISA of the guest is different from that of the host, the emulation layer often incurs high overhead. Dynamic Binary Translation (DBT) [2] is commonly used in modern emulators to greatly speed up emulation [3]. Other than enabling fast cross-ISA emulation, DBT techniques have also been widely used in dynamic binary instrumentation [4], dynamic binary optimization [5] and architecture/micro-architecture simulations [6].

DBT in QEMU focuses on simple and fast code translation to minimize translation overhead. The Tiny Code Generator (TCG) in QEMU performs few optimizations to meet the fast translation requirement. However, for extended application runtimes, improved code quality is certainly worth pursuing. For example, some recent research [7] proposed leveraging the LLVM optimization framework to improve the code quality of QEMU, achieving significant acceleration for benchmarks and applications.

SIMD (Single Instruction Multiple Data) instructions are an essential extension to most modern microprocessor architectures. For example, MMX/SSE/AVX instructions on Intel x86 architecture, and NEON on ARM and MSA on MIPS, are all SIMD instruction extensions. SIMD can use one instruction to compute multiple data, completing in a single cycle work that originally required multiple cycles. In addition to this performance advantage, it also reduces energy consumption since it avoids repeated instruction fetch and decode processes, and also saves energy in writing results. Many embedded applications can benefit from SIMD instructions, such as multimedia content processing. However, SIMD code translation has not attracted much attention in QEMU. The current QEMU requires a sequence of scalar instructions to emulate a guest SIMD instruction even if the host machine has the proper SIMD instructions to be used.

In this paper, we enhance the TCG in QEMU with improved SIMD code generation by effectively using the SIMD support in the host machines. The key is to enhance the IR in TCG with SIMD variations, referred to here as vector IR. Adding vector IRs in DBT is not trivial since there are diverse SIMD variations in different architectures. For example, Intel's SSE is register-memory based architecture, but ARM's NEON is register-register based architecture. Intel's SSE supports double precision floating point computation, while ARMv7 NEON does not.

We have designed and implemented our vector IRs in the TCG code generator of QEMU. It can support multiple different guest architectures, including ARM NEON and Intel SSE. Preliminary results running SPEC CFP2006 and the Linpack benchmark show significant speed improvements over the original QEMU code generator.

To summarize, this paper makes the following contributions:

1) We address SIMD code generation issue in the popular system emulator QEMU. We have added a set of vector IRs to TCG and corresponding backend for x86-64 to generate SSE instructions to emulate NEON instructions from ARM binaries or SSE instructions from IA32 guest binaries.

2) Our implementation was evaluated using SPEC 2006 CFP and the Linpack benchmark. Both ARM guest binaries

978-3-9815370-4-8/DATE15/©2015 EDAA

and IA32 binaries can be emulated much faster with our SIMD code generation.

The rest of the paper is organized as follows: Section 2 gives the design considerations of vector IR. Section 3 describes the implementation in QEMU. Section 4 describes the experimental settings and results. Section 5 briefly discusses related work and Section 6 concludes.

II. Design issues of Vector IR

A. HIR or LIR

Intermediate representation in a translator can include various levels such as HIR (High Level IR) and LIR (Low Level IR) [8]. LIR is closer to machine code; while HIR is closer to the source language. In a DBT, the source language is just another ISA machine code. So the question here is whether it is better to have the IR closer to the guest binary or closer to the host (or native) binary. In this work, we consider the following:

IR in a retargetable code generator: TCG in QEMU is actually a retargetable code generator which can generate code for multiple target machines, including x86, ARM, Alpha, Itanium, MIPS, PowerPC and so on. For a retargetable code generator, one approach is to generate a very low level IR tree, and use IR tree pattern matching to select instructions. The pattern matching process works like a bottom up parsing: When a pattern is matched an instruction is generated. This approach is time consuming, and not suitable for a DBT where translation time is part of the runtime. If LIR is not lower than ISA machine instructions, then LIR is not ideal since it is supposed to be architecture-neutral, and not too close to one specific machine.

IR optimization: If LIR is used, some guest binary information may be lost during the lowering process. This could limit optimizations to a higher IR level.

With the above considerations, we maintain our vector IR closer to the guest binary to preserve more information and avoid excessive translation time.

B. Vector IR format

LLVM [9] introduced a set of vector IRs. LLVM IR uses the data type to distinguish a vector IR from other scalar IRs. Consider the *Add* instruction, for example:

<result> = add <ty> <op1>, <op2>

The <ty> here is the type of operands and result. For the LLVM vector type, it consists of a size (number of elements) and an underlying primitive data type, such as <4 x i32>.

To conform to the current QEMU DBT structure, the design of our added vector IR differs from the LLVM IR. Our vector IR directly embeds its operation into opcode. Thus the LLVM IR **"%1=add<4xi32><%2>,<%3>"** will become **vadd_i32_128 r1, r2, r3**.

III. Implementation in QEMU

In general, the QEMU's DBT first translates guest instructions to TCG IR, and then translates these TCG IRs to host instructions. However, when the guest instruction is a SIMD instruction, translation flow is translated to several helper function calls, thus reducing performance while translating SIMD instructions. Therefore, our goal is to enhance the DBT of QEMU to generate respective host SIMD instructions. Our implementation includes multiple frontends such as ARM and IA32, and one backend: x86-64.

A. A simple implementation of SIMD code generation using existing helper functions

A simple approach to take advantage of host SIMD instructions is to use the SIMD intrinsic functions to implement the helper function. Existing helper functions in QEMU for the host machine are implemented with scalar instructions. Although this approach could enable the use of SIMD instructions on the host to emulate SIMD instructions of the guest binary, the helper function call would incur significant overhead. Our measurement shows the call overhead even exceeds the time to execute the SIMD instructions in the function body. To avoid such excessive overhead, we adopted the vector IR approach where SIMD instructions can be directly generated instead of indirectly through the helper function calls.

B. Implementation with vector IR

TCG IR is organized using two buffers, one for the IR's opcode and the other for the IR's parameters. The parameters put in the buffer is actually the register's number. Our vector IR is similar to the original TCG IR, but the parameters buffer we put is the offset of the guest CPUState and guest SIMD registers, rather than the register's number. Thus, we can simply and efficiently perform address calculations when generating guest SIMD instructions.

IV. EXPERIMENTAL RESULTS

In this section, we first look at our Experimental environment. We then show the performance enhancement after applying the changes to TCG in QEMU.

A. Experimental environment

We use SPEC 2006 CFP and Linpack as our benchmarks. The ARM board used in this experiment is the Samsung Exynos 5250 Arndale Board running Ubuntu 12.04 LTS. For the ARM to x86-64 emulation, the host machine is an Intel i5 running Ubuntu 13.04 LTS. For the IA32 to x86-64 emulation, the host machine is a Xeon(R) CPU E5-2620 running Ubuntu 12.04 LTS. The ARM cross-compiler used here is arm-linux-gnueabihf-gcc-4.8 and the x86 compiler used in the experiment is gcc-4.8

In the following subsection, we use"-O3" together with other auto-vectorization flags to compile benchmarks into executables. Some benchmarks are missing from the results due to compilation failures from the ARM cross-compiler, or from failures during QEMU emulation. The IA-32 executables would cause segmentation fault while the benchmark is **wrf**, and its performance result is thus omitted in these figures. The ARM-v7 NEON does not support double precision floating point arithmetic operations, so many double precision benchmarks generated no NEON instructions. Without NEON instructions in the guest binary, our enhancements will have no impact. Therefore, we replaced the double precision data

type declared in some SPEC 2006 CFP with a single precision data type. Such benchmarks are marked with the suffix _sfp. For instance, zeusmp_sfp is the single precision version of zeusmp.

B. Performance of our enhancements

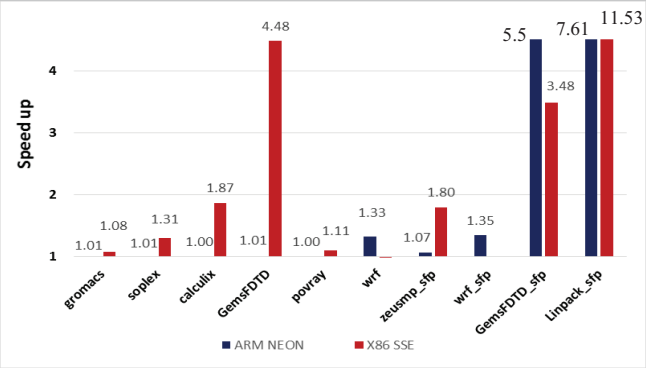

Figure 1. Performance speedup after applying the proposed approach

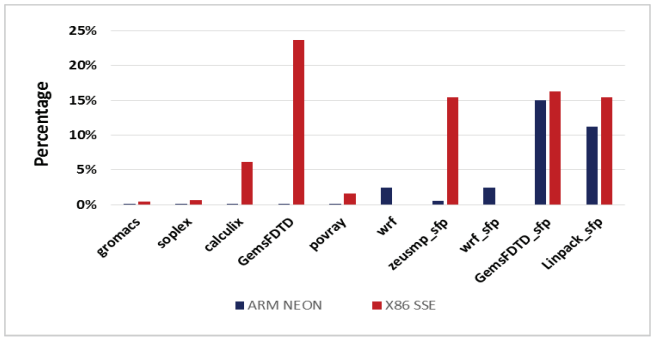

Figure 2. Execution percentage of data processing SIMD instructions

Using vector IR and host SIMD code generation, Fig 1. shows the accelerated IA32 to x86-64 emulation (red bars marked as x86 SSE). For example, Linpack runs 11.5x faster, GemsFDTD runs 4.48x faster, and many others see speed improvements between 30% to 80%. For the ARM to x86-64 emulation (blue bars marked as ARM NEON), only three programs achieved good speed improvements: Linpack, 7.6x, GemsFDTD_sfp, 5.55x, and wrf, 1.35x. ARM guest binaries do not benefit significantly because of a lack of sufficient NEON instructions in the guest binaries. Intel SIMD instructions have been on the market over two decades, and SSE code generation is well studied and its compiler support is more mature. NEON code generation, on the other hand, seems to have much room left for improvement. As shown in Fig. 2, when the ARM guest binaries contain NEON instructions, and where wrf, GemsFDTD and Linpack show a good fraction (5% to 15%) of SIMD instructions, our SIMD code generation produces significant speed improvements in QEMU.

Figure 2 reports the percentage of SIMD instructions in our benchmarks. GemsFDTD_sfp and Linpack_sfp have the highest percentage of SIMD instructions, which conforms to expectations, since they benefit most from our SIMD code generation improvement in QEMU.

In Figure 1, it is surprising to observe 11.5x and 7.62x speed improvements, respectively from Linpack for the x86 SSE and ARM NEON guest binary. The benefit from the current SIMD instruction should be bounded by 4x (i.e., the SIMD width is 4). In-depth investigation reveals the difference is due to the fact that QEMU uses software helper functions to emulate floating point instructions. Our SIMD backend emits hardware floating point instructions and thus runs much faster than the original QEMU in that it combines the speed improvement from both SIMD and the hardware floating point instructions. QEMU does not generate hardware floating point instructions even if the host machine is equipped with floating point hardware. The observed speedup seems to result in an unfair comparison, thus we modified QEMU to enable true hardware floating point instruction generation to ensure a fair comparison. We modified the helper functions used in QEMU to translate guest floating point instructions. The result of the fair comparison is shown in Figure 3.

C. Comparison with hardware-floating-point QEMU with ARM frontend and x86-64 backend.

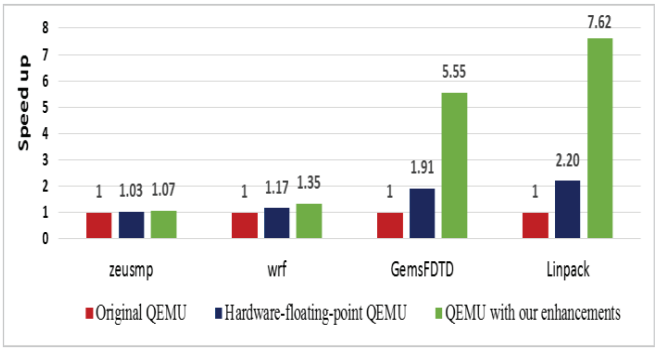

Figure 3. Performance speedup comparing the original, hardware-floating-point and QEMU with our enhancements.

In Figure 3, speed improvements are measured against the original QEMU (red bars). The performance of QEMU with true hardware floating point instruction generation is shown in blue bars. As can be observed, using hardware FP instructions, QEMU can be 1.03x to 2.20x faster than that using software helper functions. Our enhanced QEMU with SIMD code generation is still much faster than the QEMU with hardware FP instruction generation, but now within a reasonable range. For example, Linpack is now 3.46x faster, and GemsFDTD is 2.90x faster than the QEMU using hardware floating point instructions. Because the ratio of NEON instructions is rather low, the speed improvement for zeusmp and wrf is not as apparent as for GemsFDTD and Linpack.

V. RELATED WORK

The Android emulator is a critically important tool used for the development of embedded software. QEMU is the core technique in the Android emulator, and its fast emulation capability is based on DBT. In the past, QEMU development has focused on fast and accurate binary translation to minimize translation overhead. In recent years, faster versions of QEMU have been proposed such as CoreEMU [10], PQEMU [11], and HQEMU. CoreEMU and PQEMU are designed to exploit parallel emulation of virtual CPUs on host machines with multi-core processors. HQEMU focuses more on improving the quality of generated code. SIMD code generation in QEMU has a high potential of return since the speed improvements from using host SIMD instructions could be very attractive. Embedded applications are likely to drive the increased use of SIMD instructions due to the requirements of multi-media content processing. However, little research has focused on SIMD instruction translation in QEMU DBT. Michel et al. [12] proposed adding vector IR to QEMU, but their work is incomplete, and is only concerned with the ARM frontend. Li et al. [13] sought to optimize SIMD code in DBT, but their work focuses on SIMD register mapping rather than general SIMD code generation. They proposed a SIMD data type tracking algorithm to trace the type of data element in SIMD registers. By tracking the type of SIMD register, the number of data movement can be decreased. Their work was implemented in IA-32 EL.

VI. CONCLUSIONS AND FUTURE WORK

QEMU is a system emulator which has been widely used for embedded software development. The core technique used for fast emulation in QEMU is Dynamic Binary Translation (DBT), but DBT development in QEMU has largely overlooked SIMD code generation. In the current QEMU, a guest SIMD instruction is often emulated with scalar instructions even when the host machine has SIMD instructions available. This work introduces new vector IRs to the TCG in QEMU and implements a backend to turn such vector IRs into SIMD instructions on the host machine. A prototype of this enhanced TCG was implemented on both ARM and IA32 frontends, resulting in speed improvements for many SPEC 2006 CFP benchmarks on both platforms. However, because ARM binaries do not have sufficient NEON instructions in the first place, the resulting speedup was not as great as that for the IA32 binaries.

One challenge for SIMD translation in DBT is migrating legacy binaries to new machines. There is a trend for new machines to have wider SIMD instructions. For example, from SSE (128 bit) to AVX (256 bit), the SIMD length has been doubled. When translating legacy binaries with narrower to wider SIMD instructions, the challenge is to manage loop control instructions and enforce SIMD load/store alignment requirements. Future work will focus on this issue in DBT.

ACKNOWLEDGMENT

This work is supported by the Institute of Information Science Academia Sinica, Taiwan, and MOST (Ministry of Science and Technology) Taiwan.

REFERENCES

[1] Bellard, Fabrice. "QEMU, a Fast and Portable Dynamic Translator." USENIX Annual Technical Conference, FREENIX Track. 2005

[2] R.L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks and S. G. Robinson, “Binary translation”, Communications of the ACM, Volume 36 Issue 2, Feb. 1993

[3] Smith, Jim, and Ravi Nair. *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[4] Nicholas Nethercote and Julian Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation”, ACM SIGPLAN Notices - Proceedings of the 2007 PLDI conference, 2007

[5] Vasanth Bala, Evelyn Duesterwald and Sanjeev Banerjia, “Dynamo: a transparent dynamic optimization system”, PLDI '00 Proceedings of the ACM SIGPLAN 2000 conference, 2000

[6] Bob Cmelik, David Keppel, “Shade: A Fast Instruction-Set Simulator for Execution Profiling”, 94 Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems, Pages 128-137, 1994

[7] Hong, Ding-Yong, et al. "HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores." *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012.

[8] Steven S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[9]Lattner, Chris, and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004.

[10]Wang, Zhaoguo, et al. "COREMU: a scalable and portable parallel full-system emulator." *ACM SIGPLAN Notices* 46.8 (2011): 213-222.

[11]Ding, Jiun-Hung, et al. "PQEMU: A parallel system emulator based on QEMU."*Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*. IEEE, 2011.

[12] Michel, Luc, Nicolas Fournel, and Frédéric Pétrot. "Speeding-up SIMD instructions dynamic binary translation in embedded processor simulation." Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011. IEEE, 2011..

[13] Li, Jianhui, et al. "Optimizing dynamic binary translation for SIMD instructions." Proceedings of the International Symposium on Code Generation and Optimization. IEEE Computer Society, 2006.