國立交通大學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

在 HQEMU 系統模擬器的動態二元翻譯引擎上產生 SIMD 指令

SIMD Instruction Generation in the DBT Engine of the HQEMU System Simulator

研 究 生：李柏舉

指導教授：徐慰中 教授

中 華 民 國 101 年 9 月

在 HQEMU 系統模擬器的動態二元翻譯引擎上產生 SIMD 指令

SIMD Instruction Generation in the DBT Engine of the HQEMU System Simulator

研 究 生：李柏舉　　Student：Bo-Jyu Lee

指導教授：徐慰中 博士　　Advisor：Dr. Wei-Chung Hsu

國 立 交 通 大 學

資訊科學與工程研究所碩士論文

A Thesis

Submitted to Degree Program

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

In

Computer Science

September 2012

Hsinchu, Taiwan

中 華 民 國 101 年 9 月

在 HQEMU 系統模擬器的動態二元翻譯引擎上產生 SIMD 指令

研究生：李柏舉　　　　　　　　　　指導教授：徐慰中博士

國 立 交 通 大 學 資 訊 科 學 與 工 程 研 究 所 碩 士 班

摘　　要

本篇論文是基於 HQEMU 系統模擬器架構上去設計並實作出一個真正能產生出前端為 Intel SSE 的指令集到真正後端硬體上的動態執行碼轉換。HQEMU 是由 LLVM 及 QEMU 組合而成的，HQEMU 針對不同的程式行為來決定要使用原本的 QEMU TCG IRs 這個轉換進程或者使用 LLVM IRs 來取代 QEMU 本來的 TCG IRs，結合 QEMU 快翻的精神與 HQEMU 做大量優化的特性。我們修改 HQEMU 的動態二元引擎始能產生真正的 SIMD 指令，並針對這種指令新增一個優化的選項叫做向量型態的狀態對應來提升轉換的效能。我們使用工業界標準的 SPEC 2006 CFP 來驗證並改進這個轉換器得到的成效。實驗結果指出，改進後的執行時間比原本 HQEMU 的時間平均可以快上 1.35 倍。

SIMD Instruction Generation in the DBT Engine of the HQEMU System Simulator

Student: Bo-Jyu Lee Advisor: Dr. Wei-Chung Hsu

Degree Program of Computer Science
National Chiao Tung University

ABSTRACT

This thesis is to enhance the DBT engine of the HQEMU system emulator so that it can efficiently translate SIMD instructions from the target architecture into the SIMD instructions in the host machine. In the process of augmenting the DBT engine with SIMD instruction code generation capability, we also propose an optimization, called vector type state mapping for eliminate redundant SIMD load/store instructions. With the enhancement and associated optimization, we have observed 35% of speed up on average over the original HQEMU when emulating the SPEC 2006 CFP benchmarks in x86-32 binary on the x86-64 host.

誌　　謝

感謝我的指導教授，徐慰中教授，給予我指導與幫助，從教授身上學到許多寶貴的經驗與知識，和研究應抱持的態度。感謝口試委員：吳真貞教授、單智君教授、楊武教授，在口試時，給予相當多的指正與叮嚀，讓我瞭解更多東西及知道我思考不周的方向。

感謝中研院的許俊琛學長、洪鼎詠學長與陳韋任學長在我撰寫論文期間，給予我相當多的建議、幫忙與意見。感謝李原嘉同學、歐冠翬同學，在我遇到問題時，都能給我適當幫助。感謝實驗室同學詹雅淇、陳君彥，不論在計畫、課業或是論文上，都能夠互相幫助、互相鼓勵，另外，感謝在口試當天幫忙的學弟劉冠宏、傅勝于，使得口試得以進行得相當順利、流暢。最後，感謝所有幫助過我的學長姐、同學、學弟妹們，有你們的幫忙，讓我能夠有豐富的收穫，才有今日的我，感恩。

感謝金門高中 52 屆春暉社的好友們，在我撰寫論文期間的支持與鼓勵，讓我在碰到瓶頸時能夠喘口氣再繼續努力。

最後，感謝父母全力的支持，由於你們的支持，我才能繼續下一個旅程，謝謝你們。

李柏舉

2012/09/17

於國立交通大學

(Lab. 446A)

Table of Contents

摘　　要 ... i

ABSTRACT ... ii

誌　　謝 ... iii

Table of Contents ... iv

List of Tables ... vi

List of Figures ... vii

I. Introduction ... 1

II. Background and Related Work ... 3

2.1 Binary Translation ... 3

2.1.1 Static Binary Translation vs. Dynamic Binary Translation ... 3

2.1.2 Same-ISA Translator vs. Cross-ISA Translator ... 4

2.2 SIMD instructions ... 5

2.2.1 Intel's SSE ... 6

2.2.2 ARM's NEON ... 7

2.3 LLVM ... 9

2.3.1 LLVM Intermediate Representation ... 9

2.4 QEMU ... 10

2.5 Related Work ... 11

2.5.1 Speeding-up SIMD instructions via Dynamic Binary Translation ... 11

2.5.2 HQEMU ... 15

III. Design of a SIMD Code Generation Phase in the DBT Engine ... 18

3.1 Objective ... 18

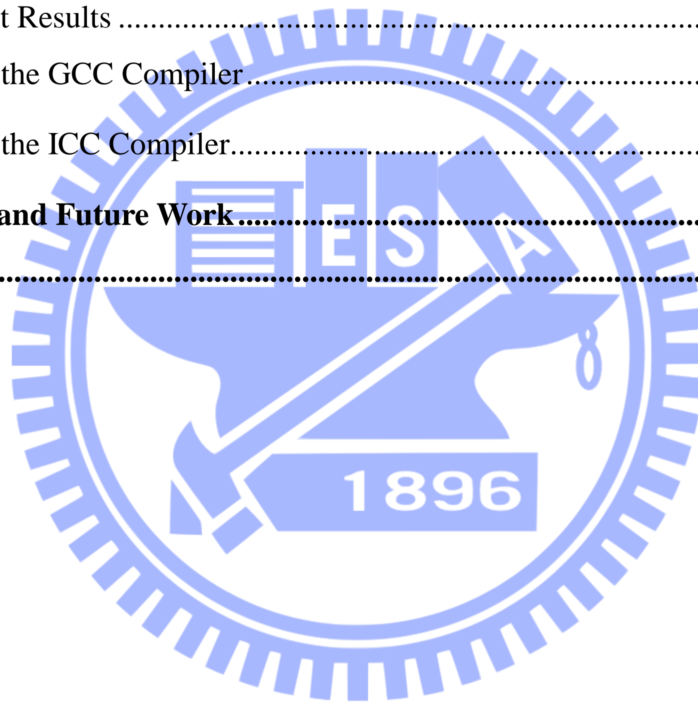
3.2 Design Issues ... 18

3.3 Using the Gcc Vector Extensions ... 20

3.3.1 Gcc Vector Extension ... 20

3.4 Using the Gcc Vector Extensions to Replace SSE Helper Function in QEMU ... 21

3.4.1 Arithmetic and Logic instructions 23
3.4.2 Shift instructions 25
3.5 Working with HQEMU 25
3.6 Optimization: Vector Type State Mapping on HQEMU 27
IV. Sanity Check Tests 31
4.1 Experimental Environment 31
4.2 Experiments Results 31
V. Experiments and Results 33
5.1 Experimental Environment 33
5.2 Experiment Results 34
5.2.1 Using the GCC Compiler 34
5.2.2 Using the ICC Compiler 37
VI. Conclusions and Future Work 43
References 44



List of Tables

Table 1. Mapping between left shift instructions 14

Table 2. Some examples of classified SSE instructions and the GCC vector extensions supported 22

List of Figures

Figure 1. SISD and SIMD 6

Figure 2. Intel's SSE registers and Scalar SIMD *addss* and Vector SIMD *addps* operation... 7

Figure 3. ARM's NEON "Packed SIMD" 8

Figure 4. Machine adaptable dynamic binary translation process 10

Figure 5. Direct mapping between *vadd. i16* NEON instruction and *paddw* SSE instruction 12

Figure 6. The *vsra* Neon instruction is translated into two TCG micro-operations 13

Figure 7. The left shift vector IR is translated into multiple SSE instructions 14

Figure 8. Mainly difference between QEMU and HQEMU 15

Figure 9. The architecture of HQEMU's DBT system on a multi-core platform........ 16

Figure 10. An example of translating *add* instruction in QEMU 19

Figure 11. An example of translating *addps* instruction in QEMU 19

Figure 12. An example of using the Gcc vector extensions with SSE 21

Figure 13. Fixing mis-alignment problems of SSE load/store in QEMU 23

Figure 14. Function implementation and differences between method 1 and method 2 24

Figure 15. (a): Original Gcc Vector Extension (b): Add Smallest Units Vector for HQEMU 26

Figure 16. (a): QEMU Version (b): Gcc Vector Extension Version (c): Modified for HQEMU 27

Figure 17. An example of doing vector type state mapping........ 28

1896

Figure 18. The process of doing state mapping at the LLVM IR stage 29

Figure 19. Host code generated from HQEMU with vector type state mapping 30

Figure 20. Speedup of QEMU-Vector compared to QEMU-Ori 34

Figure 21. Speedup of HQEMU-Vector compared to HQEMU-Ori.................................... 35

Figure 22. Proportion of SSE instruction at static time, using GCC 36

Figure 23. Proportion of SSE instruction at dynamic time, using GCC 36

Figure 24. Speedup of QEMU-Vector compared to QEMU-Ori 37

Figure 25. Speedup of HQEMU-Vector compared to HQEMU-Ori.................................... 38

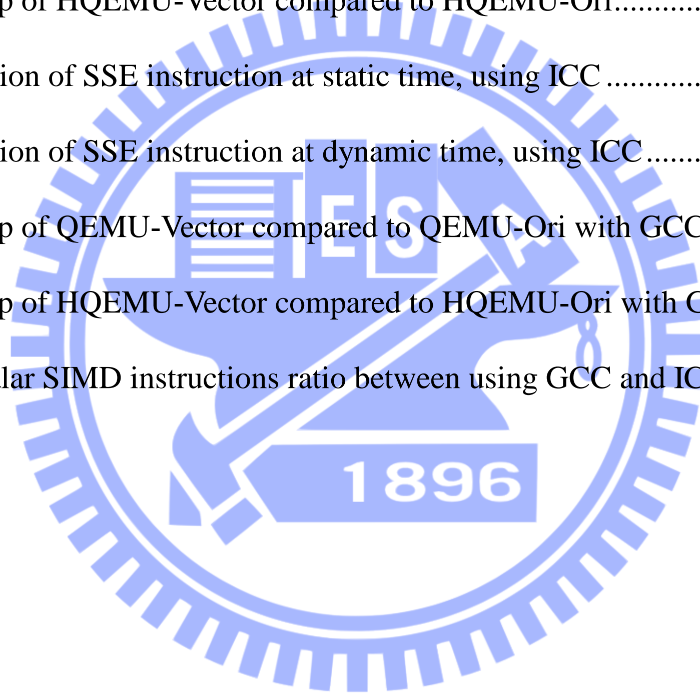
Figure 26. Proportion of SSE instruction at static time, using ICC 39

Figure 27. Proportion of SSE instruction at dynamic time, using ICC 39

Figure 28. Speedup of QEMU-Vector compared to QEMU-Ori with GCC and ICC........... 40

Figure 29. Speedup of HQEMU-Vector compared to HQEMU-Ori with GCC and ICC..... 41

Figure 30. The scalar SIMD instructions ratio between using GCC and ICC 42



I. Introduction

Audio, video and communication applications are the core activities of embedded systems. A trend of adding SIMD style instructions, such as MMX, SSE, and AVX, to the CPU in order to enhance media processing has been available on desktop computers for years. Now this trend has also found its way to embedded processor architectures such as the NEON extension on ARM and the MDMX extension on MIPS. Furthermore, such SIMD-style instructions are perfectly suitable for speeding up Floating Point computations. They are similar to vector instructions in supercomputers, except that their length of vector is much shorter than vectors in supercomputing. As such SIMD style media extensions are commonly available, many software applications contain such instructions.

Dynamic binary translation is one commonly used technique to speed up whole system simulation, such as QEMU [1], or legacy application migration such as the Rosetta and IA32-EL. With the increasingly popular use of SIMD-style instructions, how to translate SIMD instructions dynamically has attracted much attention. Effective translation of SIMD instructions is critical in such dynamic binary translation systems. In this work, we focus more on the SIMD translation in QEMU-based simulation tools. This is because QEMU is a very widely used system simulator for embedded systems, and the current QEMU does not have adequate support for SIMD instruction translation.

The purpose of this work is to come up with a solution applicable to retargetable dynamic binary translation systems that can make effective use of the SIMD computing power of the host computers. In the current QEMU, a SIMD instruction in the guest architecture is not translated into a respective SIMD instruction available in the host architecture. Instead, it is translated to call a helper function. Usually, such helper functions for the host architecture are implemented using scalar instructions rather than SIMD instructions.

We replace such helper functions of SIMD instruction, implemented in scalar instructions, by functions implemented in GCC vector IR (Intermediate Representation), and such functions will eventually get converted from vector IRs to SIMD instructions available on the host machine. In addition to the effective use of SIMD instructions of the host machine for simulating the SIMD instructions of the guest machine, we further implement an optimization called vector state mapping in HQEMU [2] to eliminate redundant vector load /store instructions and achieve a greater speed up.

The remainder of this thesis is as follows. In Chapter 2, we describe the background and the related work. In Chapter 3, we introduce the design and implement of the SIMD instruction generation of the DBT engine in HQEMU. In Chapter 4 and 5, we evaluate the performance of our design and implementation. Finally, Chapter 6 concludes this thesis.

II. Background and Related Work

In this section we first introduce binary translation. Then we explain what is SIMD (Single Instruction Multiple Data) instruction. After that, we give a simple overview of QEMU and HQEMU. The remaining of this section will discuss related works.

2.1 Binary Translation

Binary translation is aiming at transforming instructions of one ISA to another. This process can be carried out at two different times: offline, so called static binary translation (SBT) [3], [4], and on-line, so called dynamic binary translation (DBT) [5], [6]. DBT has been widely used in various applications, such as instruction set architecture (ISA) migrations, fast architecture simulations, runtime optimizations and binary instrumentations.

2.1.1 Static Binary Translation vs. Dynamic Binary Translation

Static binary translation translates guest binary code into host binary code. The advantage of static binary translation is that it can avoid the translation overhead at runtime. On the other hand, the static binary translation has code discovery problems and code location problems. For example, the branch target of an indirect branch will not be known at static time.

Dynamic binary translations (DBT) that can speed up the emulation of an application binary migration from one ISA to another is gaining importance. DBT has become the core technology of system virtualization, an often required system support in the new era of

cloud computing and mobile computing. DBT could also be used in binary instrumentation, security monitoring and other important applications.

However, there are several factors that could impede the effectiveness of a DBT: (1) emulation overhead before the binary translation; (2) translation and optimization overhead; and (3) the quality of the translated code. Retargetablity of the DBT is also an important requirement in system virtualization. It is highly desirable to have a single DBT to take on application binaries from several different ISAs and retarget them to host machines also in several different ISAs. This requirement imposes additional constraints on the structure of a DBT and, thus, additional overheads.

As a DBT is running at the same time the application is being executed, the overall performance of the translated binary on the host machine is thus very sensitive to the overhead of the DBT itself. A DBT could ill-afford to have sophisticated techniques and optimizations for better codes. However, with the ubiquity of the multicore processors today, most of the DBT overheads could be off-loaded to other cores. The DBT could thus take advantage of the multicore resources and become multithreaded itself. This allows it to become more scalable when it needs to take on more and more large-scale multithreaded applications in the future. For example, the DBT in HQEMU is taking such a multi-threaded approach to effectively minimize the code optimization overhead.

2.1.2 Same-ISA Translator vs. Cross-ISA Translator

When the guest ISA and the host ISA are the same, we refer the binary translator as a same-ISA translator. The purpose of this translator is to improve the performance or to instrument the binary code.

On the other hand, when the guest ISA is different from the host ISA, we refer the

binary translator as a cross-ISA translator. The purpose of this translator is to migrate an application from one hardware platform to another or to provide a virtual platform that the application can execute without specific hardware. QEMU is a whole system simulator using cross-ISA dynamic binary translation techniques.

2.2 SIMD instructions

Single instruction, multiple data (SIMD), is a class of parallel computers in Flynn's taxonomy. It refers to computers with multiple processing elements that can perform the same operation on multiple data simultaneously. Thus, such machines exploit data level parallelism. Figure 1 is a schematic diagram for SIMD processing.

Small-scale (64 or 128 bits) SIMD has become popular on general-purpose CPUs in the early 1990s. SIMD instructions can be found, to one degree or another, on most CPUs, including IBM's AltiVec and SPE for PowerPC, HP's PA-RISC Multimedia Acceleration eXtensions (MAX), Intel's MMX and SSE, SSE2, SSE3 SSSE3 and SSE4.x, AMD's 3DNow!, ARM's NEON technology, MIPS' MDMX and MIPS-3D.

Modern graphics processing units (GPUs) can be considered as very wide SIMD implementations, capable of processing thousands of words at a time.

SIMD instructions are now available on desktop PCs, servers and embedded systems. Among all the SIMD variations, Intel’s SSE and ARM’s NEON are the most popular and widely used, so this section explains a little more on Intel’s SSE and ARM’s NEON.

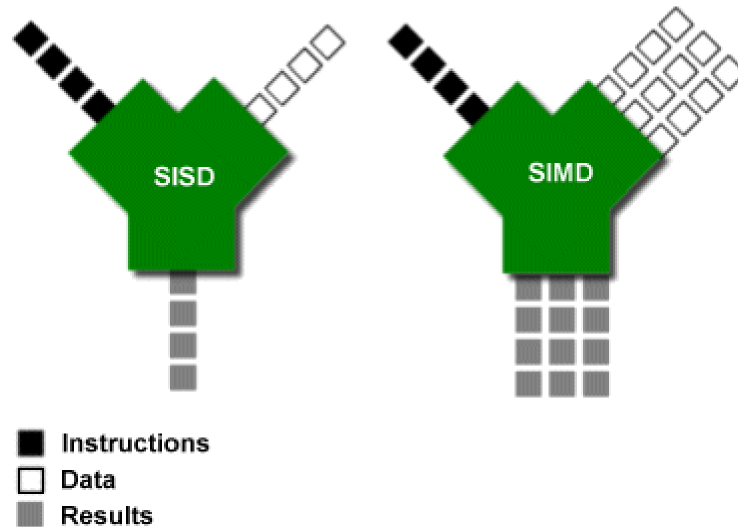
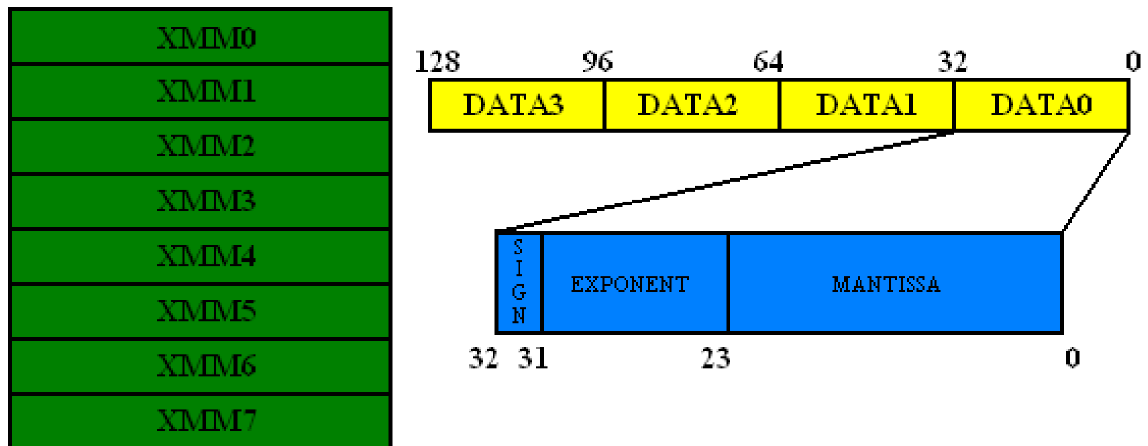


Figure 1. SISD and SIMD

2.2.1 Intel's SSE

Streaming SIMD Extensions (SSE) is introduced by Intel in 1999 in Pentium III processor. As its name implies, SSE is a SIMD instruction set. SSE instructions include four main parts: single-precision floating-point arithmetic instructions, integer arithmetic instructions, cache control instructions, and state control instructions. SSE architecture includes eight 128-bit registers, xmm0 ~ xmm7. The **xmm** registers can be used to store four 32-bit single-precision floating-point numbers or two 64-bit double-precision numbers, depending on programmer's specification. SSE instructions for FP (Floating Point) computation are different from the x87 floating-point instructions where the xmm register must be cleared with the EMMS instruction. SSE instructions can be mixed with x87 FP instructions or earlier MMX instructions, because they are using different registers. However the main drawback is that the cost of context switch would be much greater since all registers (xmm, FP, MMX) must be saved and restored. SSE has Scalar version and vector version, where the vector version is also called Packed instruction. Figure 2 is a

schematic diagram of the SSE register and an example of scalar SIMD *addss* and vector type *addps*.



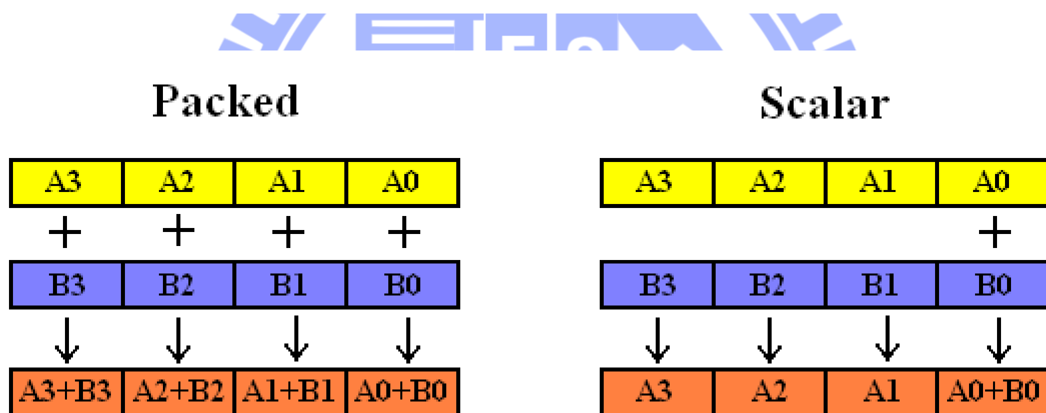


Figure 2. Intel's SSE registers and Scalar SIMD *addss* and Vector SIMD *addps* operation

2.2.2 ARM's NEON

The ARM's NEON general-purpose SIMD extension supports current and future multimedia formats. NEON instruction set is designed to accelerate multimedia and signal processing such as video encode / decode, 2D / 3D graphics, gaming, audio and speech processing, image processing, telephony, and sound synthesis, by at least 3x the performance of ARMv5 and at least 2x the performance of ARMv6 SIMD. NEON

technology is a 128-bit SIMD architecture extension for the ARM Corte-A series processors. NEON's SIMD registers can be used as 32 register with 64 bits wide or 16 registers with 128 bits wide.

NEON instructions perform "Packed SIMD" processing with the following specifics:

(1)Registers are considered as vectors of elements of the same data type.

(2)Data types can be: signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit, single precision floating point.

(3)Instructions perform the same operation in all lanes.

Unfortunately, the NEON instruction set doesn't support double precision data types and its single precision format is not fully IEEE754 compliant. Figure 3 shows one example of ARM's NEON "Packed SIMD" instructions.

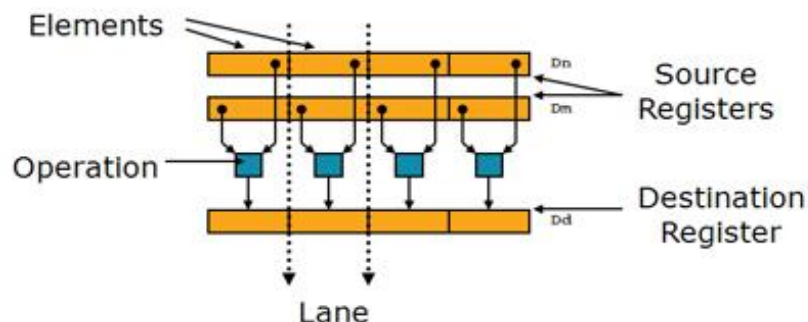


Figure 3. ARM's NEON "Packed SIMD"

2.3 LLVM

Low Level Virtual Machine (LLVM) [7] is a compiler infrastructure developed by University of Illinois. LLVM is used for optimizing programs written in arbitrary programming language during compile-time, link-time, run-time and idle-time. LLVM is also a retargetable compiler in that it can emits code for many different target machines. Since LLVM has a reliable and comprehensive optimization infrastructure, and since LLVMN is very retargetable, we would like to leverage its robust infrastructure to improve the quality of code generated in QEMU. HQEMU uses two translation pipelines to conduct dynamic binary translation: it uses the original TCG (Tiny Code Generator) in QEMU to perform quick but low quality code generation for infrequently executed portions and uses the LLVM to translate and optimize frequently execution paths. Furthermore, since LLVM is an open source project and is well-documented, it is well suited for the research community.

2.3.1 LLVM Intermediate Representation

LLVM IR (Intermediate Representation) [12] plays a central role in this process. LLVM IR has three formats serving for different purposes. The three formats are an in-memory compiler IR, an on-disk bitcode representation for fast loading by a Just-In-Time compiler or a human readable assembly language representation. LLVM provides a rich API for optimizations to be performed at runtime. All code optimizations are implemented as "LLVM IR to LLVM IR transformation passes" and code analysis is also implemented as passes, generated results can be shared between passes. We will add a new pass optimizing SIMD instructions, more detailed design will bedescribed later.

LLVM identifiers which begin with the "@'character are Global identifiers such as

functions and global variables, the remaining LLVM identifiers which begin with the '%'character are Local identifiers such as register names and types. Because the LLVM IR must follow SSA form, LLVM has unlimited number of virtual registers and each LLVM register can only be defined once. We can't use single LLVM register to represent a guest register due to the constantly changing of LLVM register, so we need a dynamic mapping table from guest CPU state register to LLVM registers.

2.4 QEMU

QEMU is an efficient and retargetable DBT system that enables both full-system virtualization and process-level emulation. QEMU is based on an intermediate representation so that the complete process of binary translation can be described in a two-phases manner, as proposed on Figure 4. QEMU can run unmodified guest operating system on the host operating system. The operating system can be X86, PowerPC, ARM or Sparc. The guest OS and the host OS can be different.

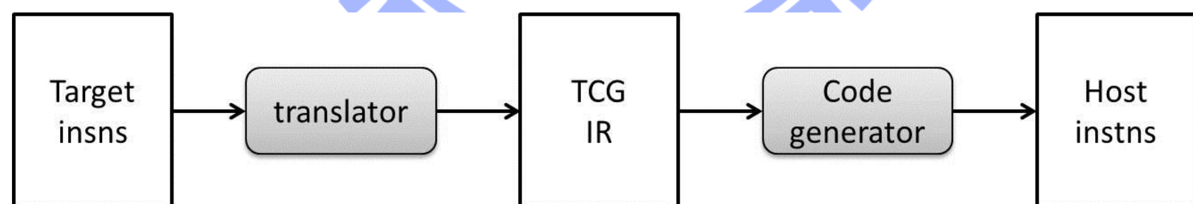


Figure 4. Machine adaptable dynamic binary translation process

QEMU uses guest basic block as a unit for translation and execution. The guest instructions in the guest basic block will be replace by several micro operations which is implemented in C programming languages, and then the micro operations will be compiled to host instructions. Therefore, there is no optimization for the guest basic block and we

need specific GCC compiler to ensure the correctness of the translation.

After QEMU version 0.10, QEMU uses tiny code generator (TCG) to parse the micro operations, which provides a small set of IR operations (about 142 operation codes) and generates the host binary code. The translation ability of TCG is still insufficient because TCG can't directly generate real host code for all guest instructions. For example, until now QEMU doesn't really support SIMD instructions because the SIMD operation will be simulated by all to scalar operations, which results in poor performance. Since most host machines have SIMD instructions available, it would be a waste not to use the host SIMD instructions to simulate the guest machine's SIMD.

2.5 Related Work

The DBT engine in the official QEMU does not translate guest machine's SIMD instructions into the host machine's SIMD instructions. In this section we will first review a related work which was about SIMD instruction generation of QEMU[8]. We also introduce HQEMU which is a derivation of QEMU with much faster simulation speed. HQEMU is the base system where we experimented with our design and implementation of a new SIMD code generation component.

2.5.1 Speeding-up SIMD instructions via Dynamic Binary Translation

In this section we describe a related study which tries to solved a similar problem as we do. This work attempts to enable the DBT engine in QEMU to emit host SIMD instructions.

The approach proposed is to add new TCG IR micro operations for SIMD instructions. The TCG in QEMU with the vector IR extensions can translate guest SIMD instructions

into the new TCG vector IR, and TCG will map the vector IRs to real host SIMD instructions . T Their goals is emulate ARM's NEON extensions with SSE on executed on an Intel Pentium based machine (guest: ARM NEON, host: Intel SSE).

Their work is based on a simple 3-addresses vector IR designed to support most existing SIMD instructions. The approach will be illustrated with concrete examples of translation from ARM NEON instruction set to Intel MMX/SSE in this section. In the subsequent examples, they divide all instructions into three translation cases. We will discuss each case and give appropriate examples.

a) One-to-one mapping between instructions:

It is the presence of an exact equivalence between a target SIMD instruction and a host SIMD instruction. The behavior of the SIMD DBT in this situation is quite similar to the one of the scalar DBT. All we have to do is to guarantee to convey operands to correct registers and retrieve the results from the correct registers. Figure 5 illustrates the translation of an ARM Neon vadd.i16 into an Intel MMX/SSE paddw.

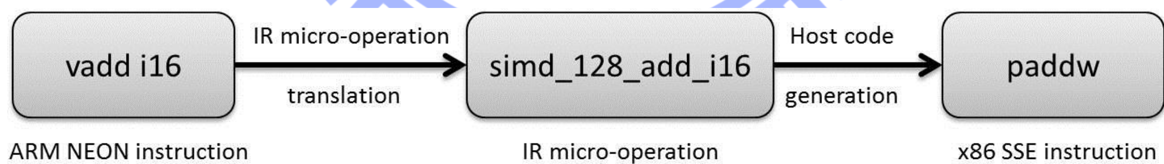


Figure 5. Direct mapping between *vadd. i16* NEON instruction and *paddw* SSE instruction

b) No direct mapping available:

There exists no direct mapping between guest SIMD instructions and the host SIMD instructions. Most of the cases are due to a lack of generality of the operations performed by the target SIMD instructions. In this case it is not very useful to have a vector IR for that

instruction. The strategy in such a case is to split the target SIMD instruction into more elementary operations available in the IR. Figure 6 gives an example of this situation with the translation of the ARM Neon vsra.u32 instruction (which is performing a right shift on operands and accumulate the shifted results in the output register) to two elementary IR micro-operations simd_128_shr_i32 and simd_128_add_i32. The code generator can then find an equivalent for each micro-operation, i.e. psrld and paddd.

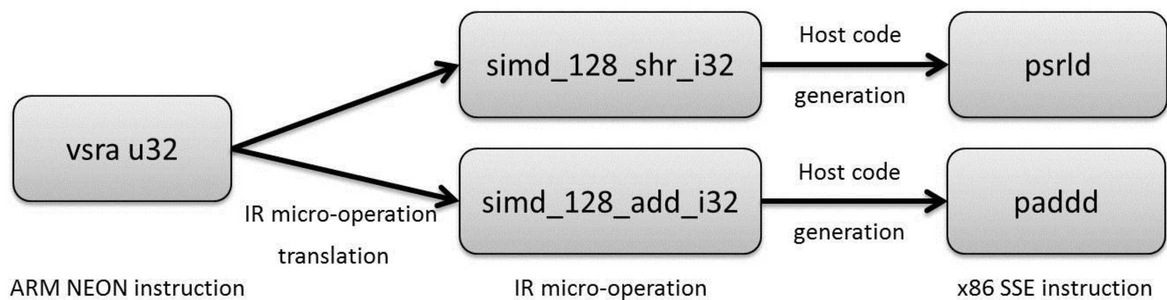


Figure 6. The *vsra* Neon instruction is translated into two TCG micro-operations

c) Exceptional cases:

This situation happens when an SIMD instruction of the target can be translated into a corresponding IR but no equivalent translation is available in the host SIMD instruction set. As shown in Table 1, all versions of the shift are available in ARM Neon SIMD instruction set. As it can be realized from this table, there exists no instruction for shifting 8 bits values. As this operation is available in all other instruction sets, it is included in the IR. The code generator has to solve this situation by generating multiple host instructions, as shown in Figure 7. The example given in Figure 7 is for the translation of an 8 bits logical left shift emulated by a 16 bits version.

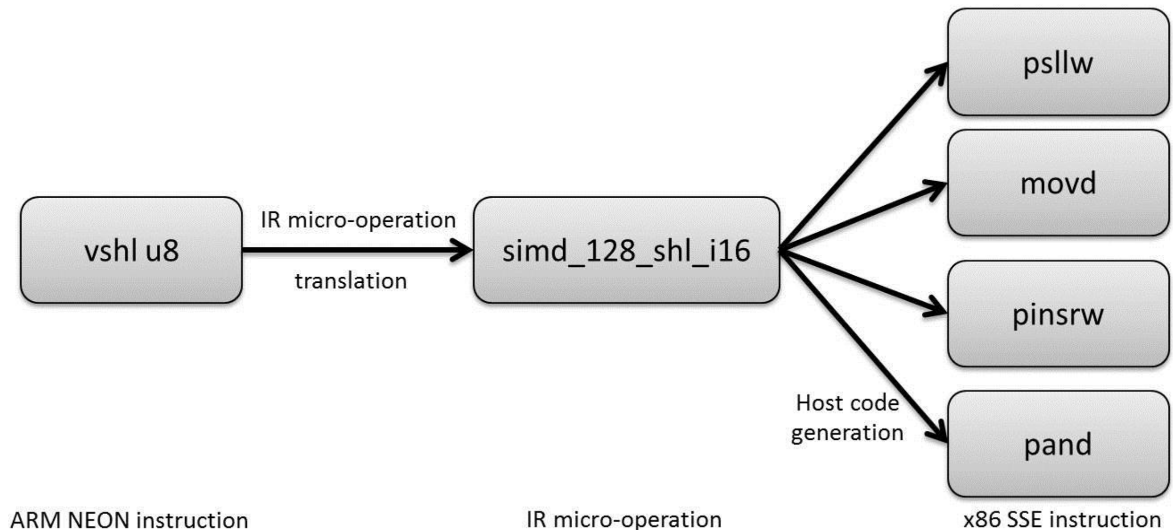


Figure 7. The left shift vector IR is translated into multiple SSE instructions

Table 1. Mapping between left shift instructions

Operation	Neon instruction	SSE instruction
shl 8 bits	vshl i8 Qd, Qm, #imm	N/A
shl 16 bits	Vshl i16 Qd, Qm, #imm	psllw xmm1, xmm2
shl 32 bits	Vshl i32 Qd, Qm, #imm	pslld xmm1, xmm2
shl 64 bits	Vshl i64 Qd, Qm, #imm	psllq xmm1, xmm2

In summary, their approach is to add the SIMD IR into TCG for different targets and hosts, then mapping new TCG vector IR to the host SIMD instruction. This approach is difficult to implement since both the front-end translation and the back-end code generation must be modified for every guest and every host machine with SIMD instructions. In our evaluation work, we choose to take a more efficient implementation which is readily compatible to the current QEMU, to get a reasonably good performance on SIMD emulation.

2.5.2 HQEMU

Hybrid-QEMU(HQEMU) is a multi-threaded hybrid DBT system, using QEMU and LLVM as building blocks. HQEMU uses QEMU exiting DBT as its frontend for fast binary code emulation, and uses LLVM, a popular compiler infrastructure with sophisticated compiler optimizations as its backend, for hot code optimization. With the hybrid QEMU (frontend) + LLVM (back-end) approach, HQEMU effectively achieves high performance emulation with good code quality and low translation overhead. Figure 8 shows the main idea of HQEMU and its difference with QEMU.

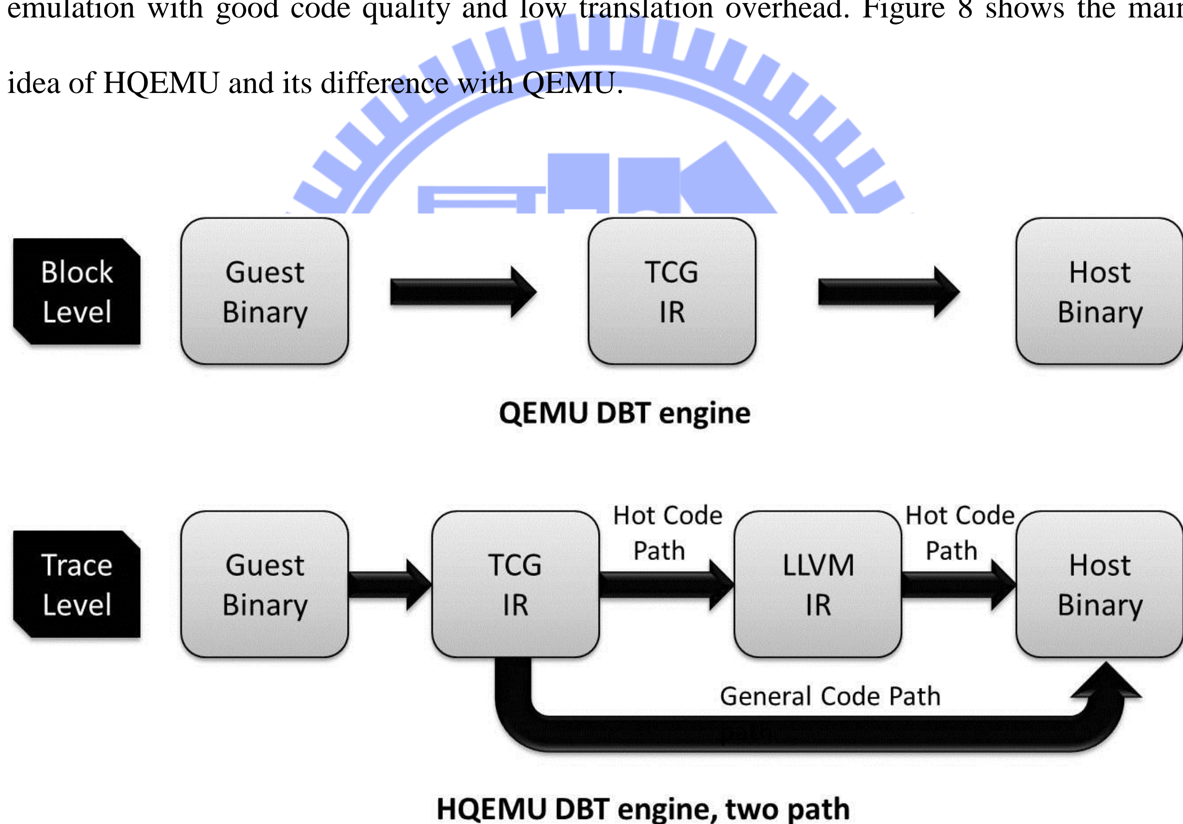


Figure 8. Mainly difference between QEMU and HQEMU

HQEMU's highlights are listed as follows:

- HQEMU develops a multi-threaded and retargetable DBT on multi-cores that achieved low translation overhead and good translated code quality on the target binary applications. This hybrid approach is good for both short-running and long-running

applications.

- HQEMU proposes a trace combination technique to improve existing trace selection algorithms. It could effectively combine/merge separated traces based on the information provided by the on-chip HPM (Hardware Performance Monitor). They demonstrate that such feedback-directed trace merging optimization can significantly improve the overall code performance.
- Experimental results show that HQEMU could improve the performance by a factor of 2.4X and 4X over QEMU, and are only 2.5X and 2.1X slower than the native execution for x86 to x86-64 emulation using SPEC2006 integer and floating point benchmarks, respectively.

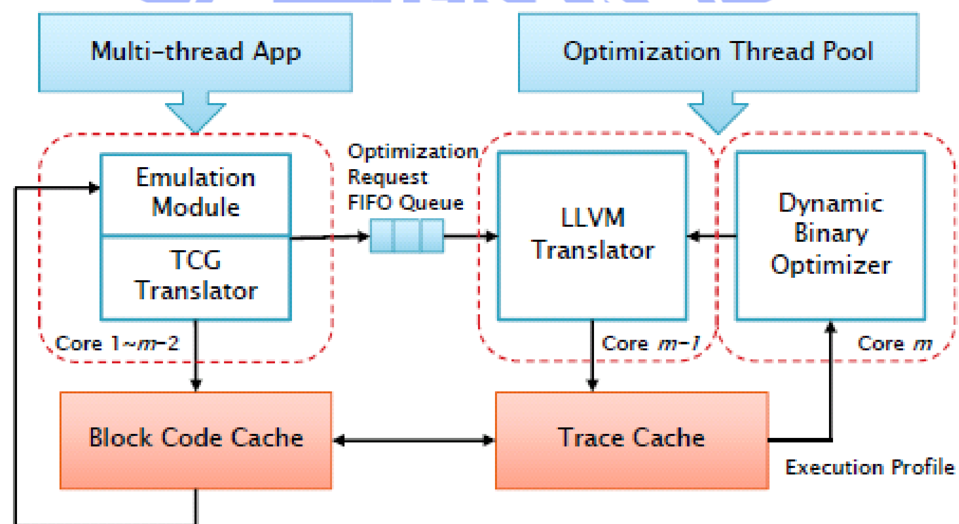


Figure 9. The architecture of HQEMU's DBT system on a multi-core platform.

Figure 9 illustrates the organization of HQEMU. It has an enhanced QEMU as its frontend, and an LLVM together with a dynamic binary optimizer (DBO) as its backend. DBO uses a HPM-based feedback-directed runtime optimization scheme. In its current implementation, QEMU is running on one thread and LLVM+DBO are running on a different thread. Two code caches: a block-code cache and a trace code cache, are built in the DBT system to store host translated binary codes with different optimization levels.

Although HQEMU adds a lot of optimizations, but there are still some places that can

be improved. For example, its current DBT engine does not generated good SIMD instructions. The issue is that when the DBT engine encounters guest SIMD code, the DBT engine will translate the SIMD instruction into scalar instructions, regardless of whether the host's hardware support for SIMD instructions. This paper is mainly to improve the SIMD code generation of the DBT engine in HQEMU.

III. Design of a SIMD Code Generation Phase in the DBT Engine

In this chapter, we first describe the problem that we observed from the SIMD code generation in the DBT engine of QEMU/HQEMU. Then we introduce our proposed solution for SIMD code generation step by step. Finally, this approach was implemented on HQEMU to demonstrate its performance. The remainder of this section describes an optimization we included to enhance the performance of generated SIMD code in HQEMU, called vector type state mapping.

3.1 Objective

Before explaining the problem, we must state our goals. The goal is to enable the DBT engine of HQEMU to generate INTEL SSE SIMD instructions with minimal modifications to the HQEMU DBT engine. Since HQEMU uses QEMU as a basis, we shall discuss the design and implementation of the SIMD code generation phase based on QEMU, and then test it on HQEMU.

3.2 Design Issues

In general, the DBT engine of QEMU will translate the guest instructions into the corresponding TCG IR and then mapping the IR's to host binary instructions. The SIMD type instructions are different in that the DBT translates them (guest SIMD instructions) into a call in TCG IR and then the call will be mapped to a function call which jumps to a helper function with scalar operations. Figure 10, and 11 shows the difference of general code

generation and SIMD code translation process in QEMU.

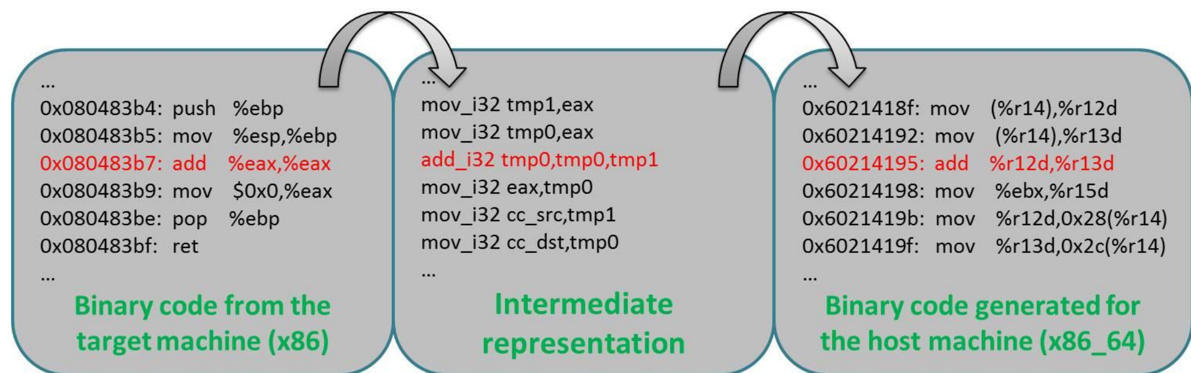


Figure 10. An example of translating *add* instruction in QEMU

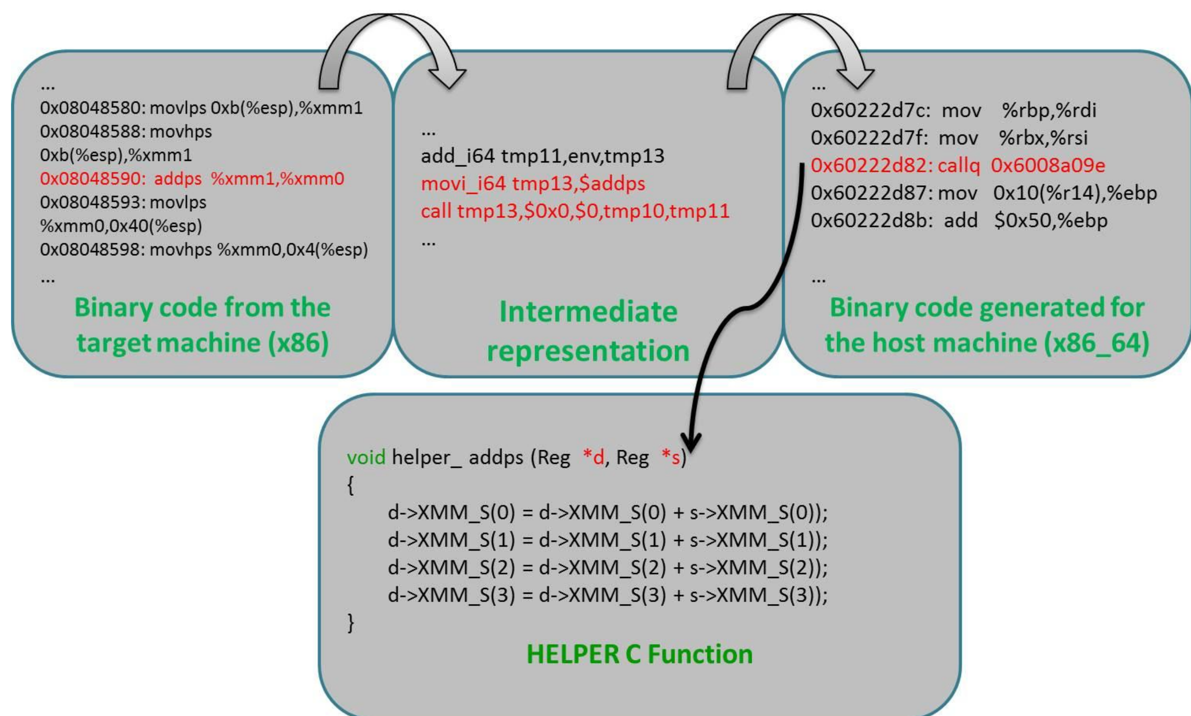


Figure 11. An example of translating *addps* instruction in QEMU

From Figure 11, we can observe that several improvements can be made to the current SIMD emulation in QEMU. Instead of splitting the 128-bit operation into four 32-bit operations, we could replace the four scalar operations with one real SIMD host instruction. Furthermore, the function call could be inlined to avoid calling overhead. The parameters can be bounded to the SIMD instruction during function inlining. Our design is trying to realize the above optimizations.

3.3 Using the Gcc Vector Extensions

In order to replace scalar operations with a SIMD operation, we adopt the Gcc vector extension to realize SIMD code generation. A more formal approach is to add new TCG IR for SIMD instructions. However, this formal approach requires more work in both the front-end and the back-end. In the front-end, a new code generator to translate the guest SIMD into the vector IR is needed. In the back-end, a code generator to convert the vector IR into the host SIMD instruction must be in place. In this work, we use the Gcc vector extension to replace the scalar instructions in the vector helper function. When the helper function is called, the Gcc vector IR will become host SIMD instruction, and the helper function call may be inlined to eliminate the calling overhead.

3.3.1 Gcc Vector Extension

The Gcc vector extension is a very powerful extensions to use SIMD code in a portable way. For example, it supports Intel SSE, ARM NEON, PowerPC AltiVec and Alpha. Gcc would choose the best possible extensions during compile time. When you compile code without SIMD options, the binary will remain compatible. The downside is that this extension doesn't allow using all the features of all SIMD code. Therefore, using this method can not completely replace all helper functions of INTEL SSE. A detailed explanation of the design will be in the next section. Figure 12 shows the example of using the Gcc vector extension with/without SSE options.

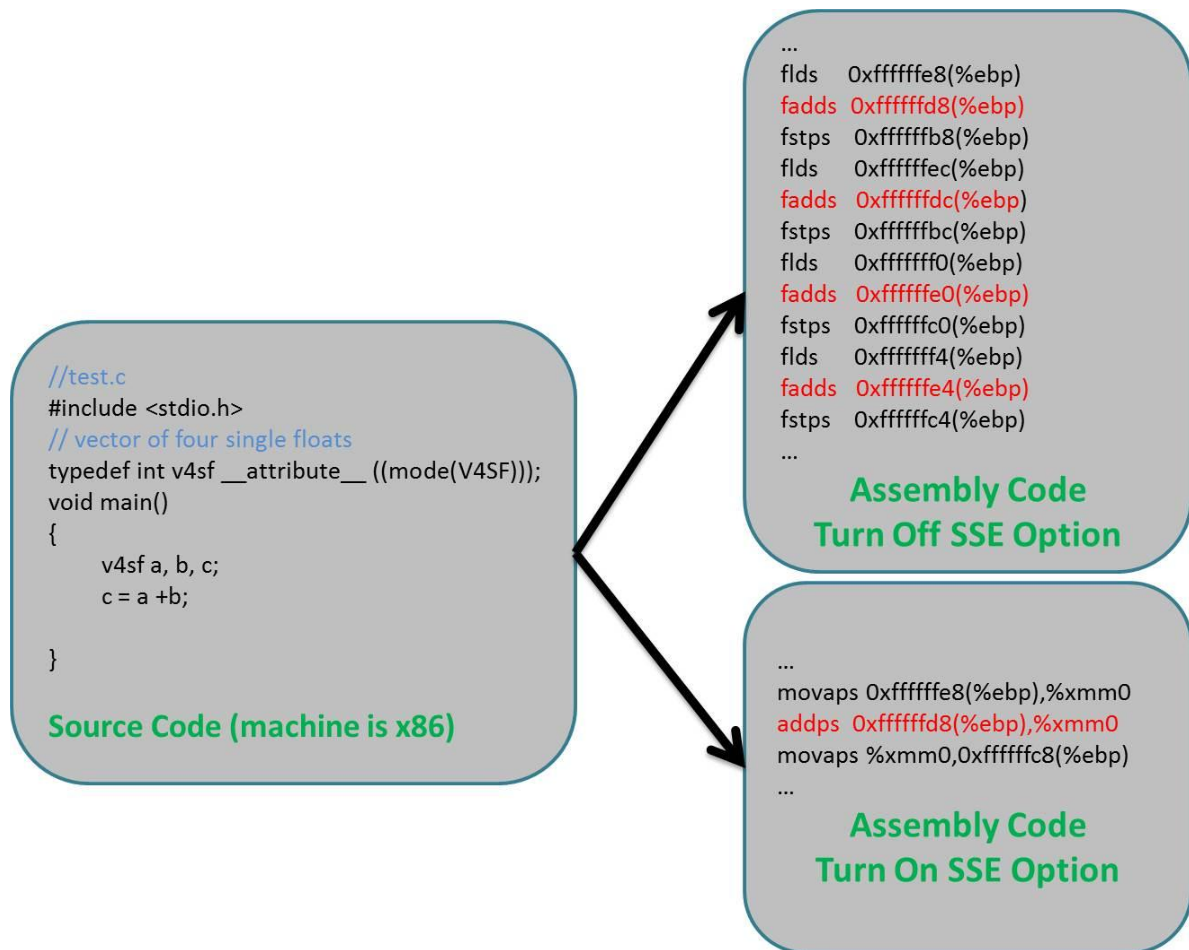


Figure 12. An example of using the Gcc vector extensions with SSE

3.4 Using the Gcc Vector Extensions to Replace SSE Helper Function in QEMU

In this work we focus on Intel SSE instruction generation. For our convenience, we classified SSE instructions into several types, which are data move instructions, data type conversion instructions, arithmetic instructions, logic instructions and other special instructions, as shown in Table 2. As we mentioned above, this extension doesn't allow using all the features of all SSE instructions. SSE has many different versions, we implement SSE2 in this work. SSE2 has 223 instructions and we select 29 instructions which are more commonly used. We use Oprofile to analysis the benchmark 410.lbm from

the SPEC2006 CFP suite, and found the most time-consuming function is LBM_performStreamCollide() where SSE instructions account for 76% of all executed instructions, and among these SSE instructions, ALU and shift SIMD instructions are responsible for half of them.. Table 2 list the Gcc vector extensions supported in our prototype system.

Table 2. Some examples of classified SSE instructions and the GCC vector extensions supported

SSE Instructions Type	Example	Gcc Vector Support
Data Move	movaps, movss …etc	No
Data Type Conversion	cvtps2dq, cvtpd2dq …etc	No
Arithmetic	addps, sqrtps …etc	Yes
Logic	or, xor, and, shift …etc	Yes
Other	paddusb, punpckhbw …etc	No

We will explain how to implement these instructions in the next section. We have added some flags into the QEMU configure file (QEMU_CFLAGS = -msse2 -mfpmath=sse) to enable Gcc to compile with SSE instruction set of the host. Then we replace the QEMU helper functions for SIMD instructions with the Gcc extensions, as well as adding appropriate cflags to execute QEMU program which might generate segmentation faults due to misaligned accesses from SIMD loads/stores. We found this bug is because of the original QEMU designers just translate SSE into scalar instruction does not take into account the alignment of executing real SSE. Figure 13 shows how our solution fixes the alignment problem of QEMU.

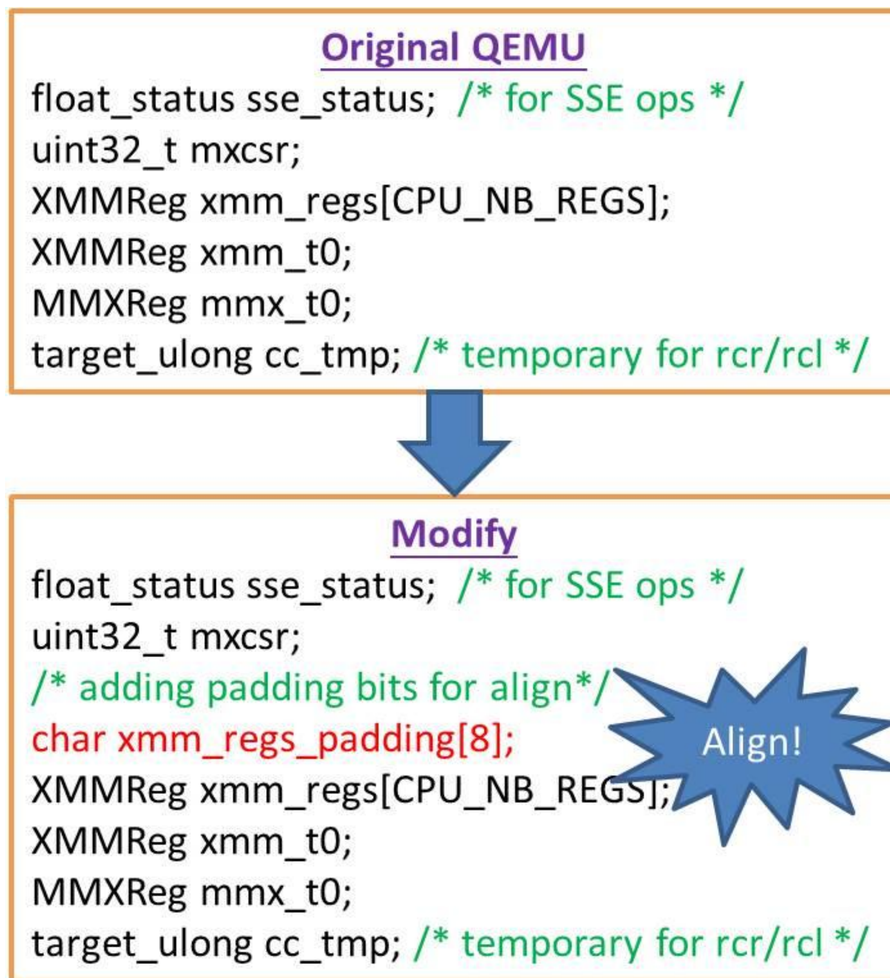


Figure 13. Fixing mis-alignment problems of SSE load/store in QEMU

3.4.1 Arithmetic and Logic instructions

The arithmetic and logic instructions are calculating the xmm registers. Most of them need two source operands to calculate the result then store to the destination. Because of these two types of instruction format are similar, we put together for explaining how they are implemented.

We will describe two methods of implementation and choose the better method to use. The first approach is obtained from the Gcc vector extension website which can be executed correctly, but not efficiently because it has excessive load/store instruction overhead. The second approach is using a casting method to avoid generating multiple load/stores , so we

select the second approach to implement. Figure 14 shows that the helper function implementation and the differences between method 1 and method 2.

```
typedef float v4sf __attribute__ ((mode(V4SF)));
union f4vector
{
    v4sf v;
    float   f[4];
    float64 df[2];
};
```

Method 1
General Usage

```
void helper_addps(Reg *d, Reg *s)
{
    union f4vector a,b,c;
    //multi-load
    a.f[0]=d->XMM_S(0); a.f[1]=d->XMM_S(1); a.f[2]=d->XMM_S(2); a.f[3]=d->XMM_S(3);
    b.f[0]=s->XMM_S(0); b.f[1]=s->XMM_S(1); b.f[2]=s->XMM_S(2); b.f[3]=s->XMM_S(3);
    //will produce one addps instruction
    c.v=a.v+b.v;
    //multi-store
    d->XMM_S(0)=c.f[0]; d->XMM_S(1)=c.f[1]; d->XMM_S(2)=c.f[2]; d->XMM_S(3)=c.f[3];
}
```

```
typedef float v4sf __attribute__ ((mode(V4SF)));
```

```
void helper_addps(Reg *d, Reg *s)
{
    // casting more efficient
    v4sf *src = (v4sf*)s;
    v4sf *dst = (v4sf*)d;
    // will produce one addps
    *dst += *src;
}
```

Method 2
Using Casting

Figure 14. Function implementation and differences between method 1 and method 2

3.4.2 Shift instructions

The shift instructions we implemented with the same casting method to replace the original QEMU helper function yield errors at the compilation stage. The problem here is due to the use of an inadequate Gcc version. The version we initially used was 4.5.2, yet the version 4.6.x is required to support SSE shift instructions. But even with version 4.6.x, this issue will still introduce some bugs when combined with HQEMU, more details will be explained in the next subsection.

3.5 Working with HQEMU

In HQEMU, the LLVM IR also supports vector type IRs, so we just convert the helper function of SSE instructions at compile time into respective LLVM IR code.

When the modified helper functions are combined with HQEMU, there have been some problems incurred. The first problem is that HQEMU can only identify the smallest units of vector instructions at the LLVM IR stage. In order to solve this problem, we must prepare 128-bits data types with smallest units before doing real operations and store back. Figure 15 shows the solution when combined with HQEMU. The second problem is that the shift instructions are only supported with Gcc versions newer than 4.6.x, but the LLVM version used was based on Gcc 4.5.2. We have to adjust the implementation of shift instructions, reserve the vector type load and store before and after the shift operator, then extract each element from the vector to do shift operations individually. However, this approach is slower than the Gcc vector extension, but is more efficient than the original QEMU. Because the implementation of QEMU is to read each element from the memory and perform shift operation and then store it back to memory. Our method does not reduce the

scalar shift operations but would avoid several load and store operations, as shown in Figure 16.

```
void helper_addps(Reg  *d, Reg *s)
{
     // casting more efficient
     v4sf *src = (v4sf*)s;
     v4sf *dst = (v4sf*)d;
    // will produce one addps
    *dst += *src;
}
```

(a) Using Gcc Vector Extension

```
void helper_addps(Reg  *d, Reg *s)
{
        //add smallest units
        v16qi* src = (v16qi*)s;
        v16qi* dst = (v16qi*)d;

        v4sf srcv = (v4sf) *src;
        v4sf dstv = (v4sf) *dst;
        dstv+=srcv;

        //store back
        *dst=(v16qi)dstv;

}
```

(b) Modified with Smallest Units

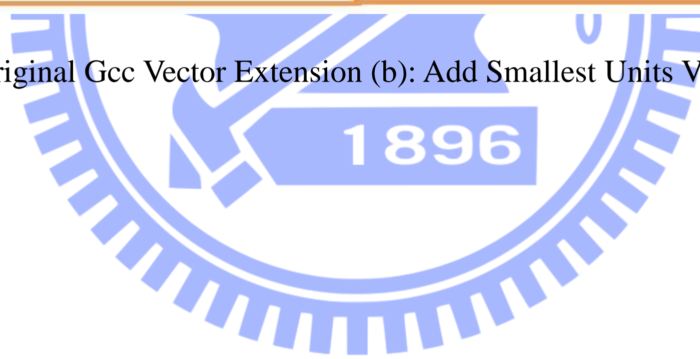


Figure 15. (a): Original Gcc Vector Extension (b): Add Smallest Units Vector for HQEMU

```
void glue(helper_psrlw, SUFFIX) (Reg *d, Reg *s)
{
    //load form memory
    //then do shift operator
    //final store back to memory
    d->W(0) = d->W(0) >> s->W(0) ;
    d->W(1) = d->W(1) >> s->W(1) ;
    d->W(2) = d->W(2) >> s->W(2) ;
    d->W(3) = d->W(3) >> s->W(3) ;
    d->W(4) = d->W(4) >> s->W(4) ;
    d->W(5) = d->W(5) >> s->W(5) ;
    d->W(6) = d->W(6) >> s->W(6) ;
    d->W(7) = d->W(7) >> s->W(7) ;
}
```

(a) Original Helper Function of Shift on QEMU

```
void glue(helper_psrlw, SUFFIX) (Reg *d, Reg *s)
{
    //load once , shift once, store one
    v8hi *dst=(v8hi*)d;
    v8hi *src=(v8hi*)d;
    *dst  >>= *src;
}
```

(b) Using Gcc Vector Extension

```
void glue(helper_psrlw, SUFFIX) (Reg *d, Reg *s)
{
    //load once
    v16qi *dst = (v16qi*)d;
    uv8hi v = (uv8hi) *dst;

    //extract
    uv8hi temp = {
        V16ui(v,0) >> shift,
        V16ui(v,1) >> shift,
        V16ui(v,2) >> shift,
        V16ui(v,3) >> shift,
        V16ui(v,4) >> shift,
        V16ui(v,5) >> shift,
        V16ui(v,6) >> shift,
        V16ui(v,7) >> shift
    };

    //store once
    *dst = (v16qi)temp;}
}
```

(c) Using Extract Individual Element from Vector

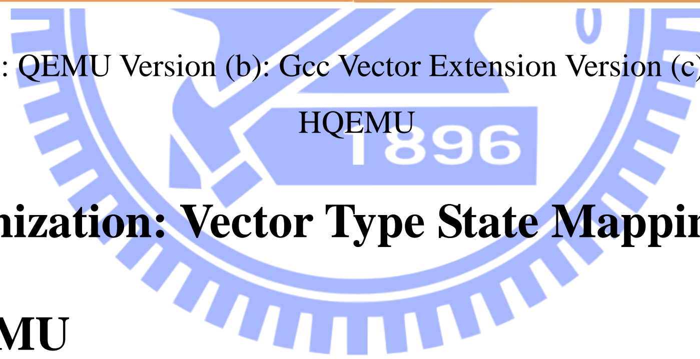


Figure 16. (a): QEMU Version (b): Gcc Vector Extension Version (c): Modified for HQEMU

3.6 Optimization: Vector Type State Mapping on HQEMU

We implemented a vector type state mapping optimization to improve the code quality for SIMD instruction execution on HQEMU. The reason why we do this optimization is because we want to eliminate the unnecessary load / store instructions. This concept is to promote some SIMD data to host xmm registers to avoid unnecessary load/stores when accessing the guest xmm registers. Figure 17 shows the example of doing vector type state mapping.

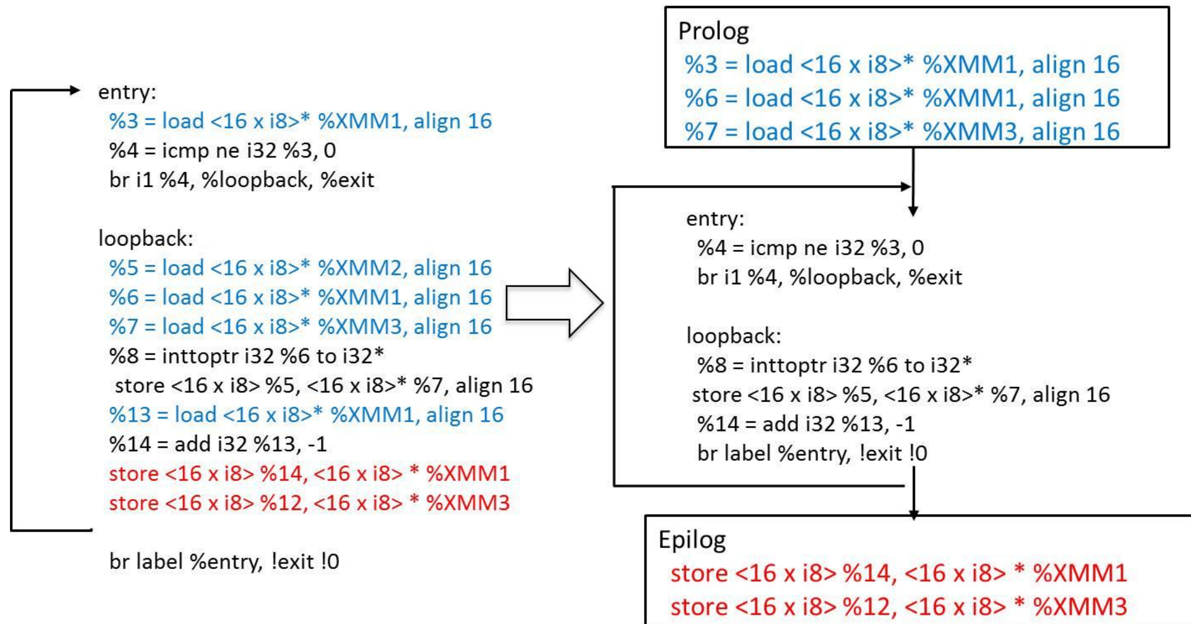


Figure 17. An example of doing vector type state mapping

We observed that the original SIMD code generated from HQEMU produced more redundant SIMD load / store instructions when translate vector type instructions. Because HQEMU only implemented state mapping for scalar type operation, vector instructions cannot enjoy the redundant load/store elimination benefit from state mapping. Therefore, we have to implement the vector type state mapping optimization to deliver the expected performance from SIMD code generation.. To achieve this purpose, we created a data structure called CPUX86State array for recording the usage of each xmm registers to determine which load / store can be merged and using the Mem2Reg pass of LLVM to complete the function of vector type state mapping. Figure 18 shows the process of implementing the vector type state mapping at the LLVM IR stage.

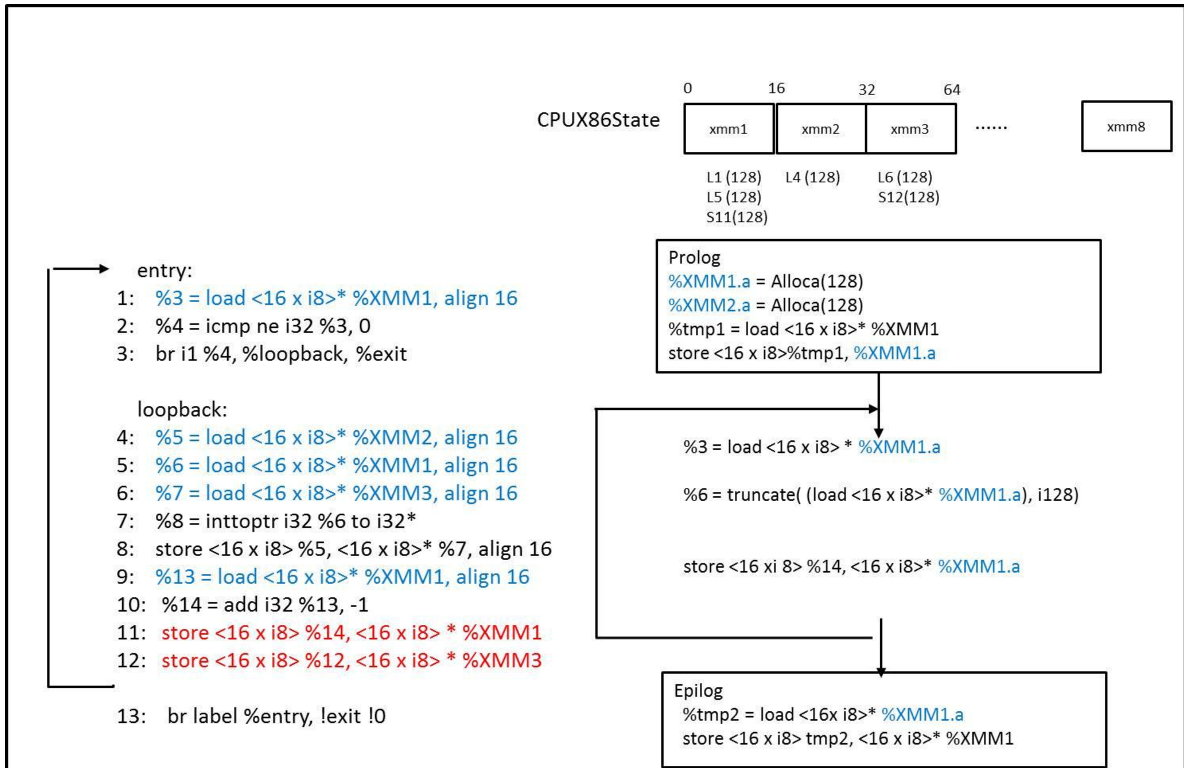


Figure 18. (a): Using CPUX86State array to help us do the mapping

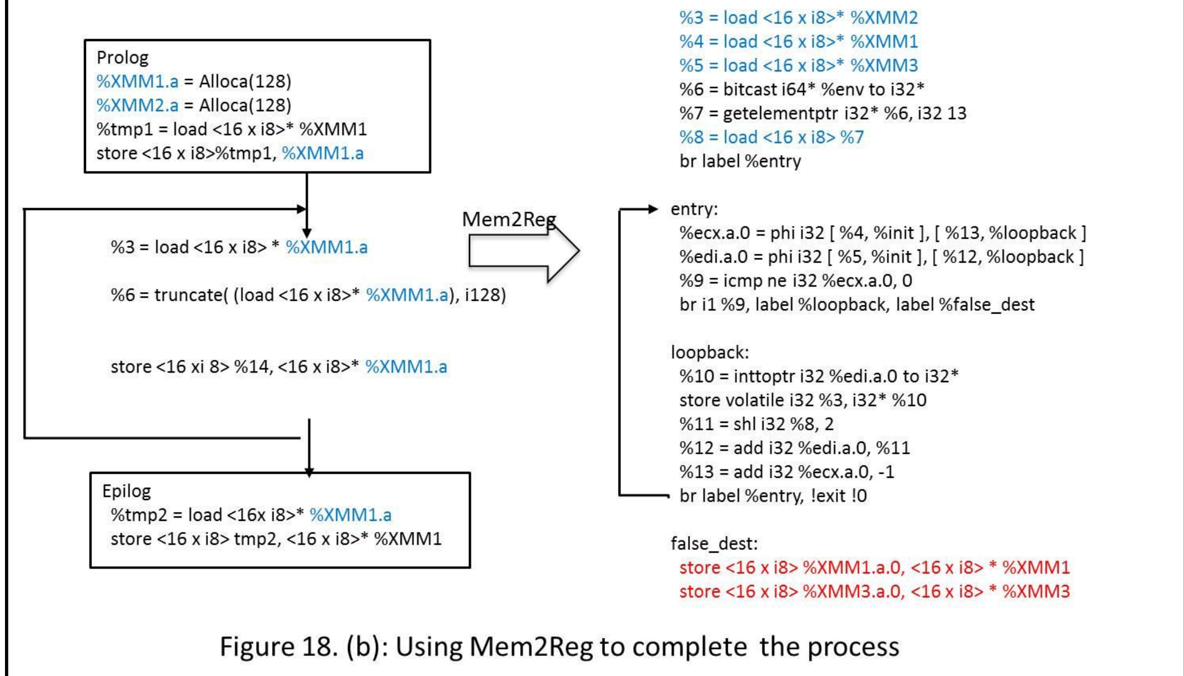


Figure 18. (b): Using Mem2Reg to complete the process

Figure 18. The process of doing state mapping at the LLVM IR stage

After the implementation of the vector type state mapping optimization, we verify that this method is correct and indeed generate better code quality. Figure 19 shows the host code generated from HQEMU with vector type state mapping optimizations.

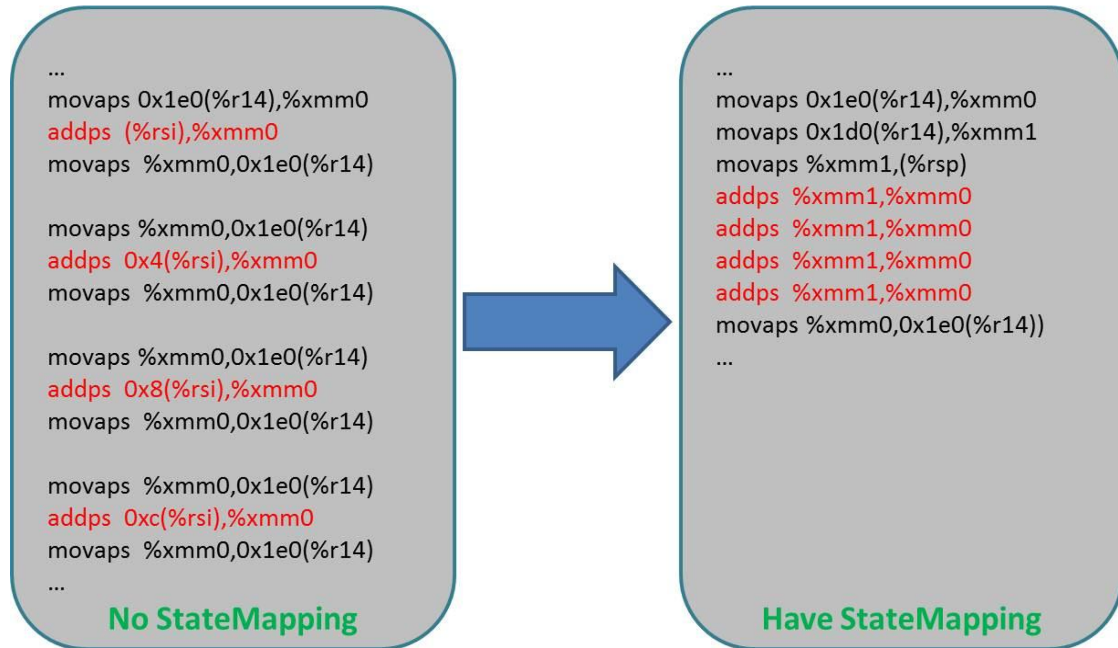
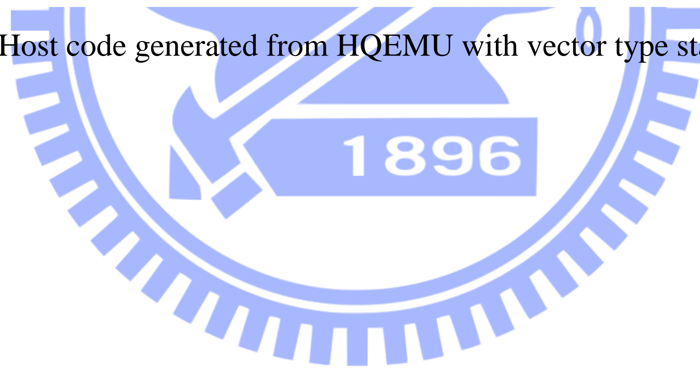


Figure 19. Host code generated from HQEMU with vector type state mapping



IV. Sanity Check Tests

In this section, we use some simple experiments (e.g. micro-benchmarks) to verify our modification for SIMD instructions generation and discuss the performance obtained. In the simple micro-benchmark test, we evaluate the performance of QEMU and HQEMU by using the Gcc vector extensions to translate the x86 front-end with SSE instruction into the x86-64 binary code. We use QEMU version 0.13.0 as the emulation engine module, and use LLVM version 2.8 to implement the translation module.

4.1 Experimental Environment

Our experiments run on an Intel Xeon CPU X5550 @ 2.67GH with 24GB RAM machine. The operating system is 64-bit Ubuntu distribution Linux. The benchmark we used in this section is a loop full of SIMD instructions. The benchmark is compiled by gcc-4.5.2 with "-msse2 -mfmath = sse" flags for QEMU. Then we compare the performance of our version called QEMU-Vector to the original QEMU.

4.2 Experiments Results

Our test bench is a loop full of *addps* SSE instruction which does four single precision floating-point additions at once. We expect this SSE instruction filled loop will have 4X speedup over the original QEMU when our SIMD code generator is used by QEMU. First, we compare the performance between QEMU-Vector and QEMU-Ori, the difference between these two versions is only the helper function is replaced. Comparing the runtime, QEMU-Vector is 1.56X times faster than QEMU-Ori. In the second step, we use HQEMU

which would inline the helper function to eliminate the function call overhead without other optimizations. This time, HQEMU-Vector is 2.86X times faster than HQEMU-Ori. This is much better than the first round where QEMU-Vector and QEMU-Ori are compared. This indicates the importance of function inlining for this SIMD code generation. However, 2.86X is still away from the ideal 4X. Why our SIMD code generation can't yield 4X acceleration? To understand the limitations, we conduct another set of analysis. We observed that SIMD instructions on x86-64 have a longer latency than an x86 scalar floating point instruction. Although the bandwidth of a SIMD instruction is 4X then scalar version, the latency is not. Whenever the instruction latency plays a role, it is difficult to achieve 4X speedup.

V. Experiments and Results

In this section, we use separate compilers (GCC and ICC) to evaluate the performance of running the SPEC2006 CFP benchmarks for our SIMD code generator. Our ultimate goal is to improve the performance of HQEMU when simulating guest binaries containing SIMD instructions. We first show the impact of our code generator approach on the original QEMU. Since our design is influenced by the features in HQEMU, so the full performance potential of our SIMD DBT engine can only be fully unlocked on HQEMU, not on QEMU. We use Gcc version 4.5.2 and Icc version 10.0 in our experiments. In HQEMU, we use LLVM version 2.8 and llvm-gcc version 4.2.1 with default optimization options.

5.1 Experimental Environment

Our experiments run on an Intel Xeon CPU X5550 @ 2.67GH with 24GB RAM machine. The operating system is 64-bit Ubuntu distribution Linux. The benchmarks we use in the experiments are SPEC2006 CFP. All benchmarks are compiled by gcc-4.5.2 with "-O3 -m32 -msse2 -mfpmath=sse -fno-strict-aliasing -ftree-vectorize" flags and icc-10.0 with default options.

We run all benchmarks via the standard SPEC runspec script with configuration files. Then we compare the performance of QEMU-Ori, QEMU-Vector, HQEMU-Ori and HQEMU-Vector. QEMU-Vector and HQEMU-Vector have our SIMD DBT engine.

5.2 Experiment Results

5.2.1 Using the GCC Compiler

In this section, we first compare the performance between QEMU and QEMU-Vector with the GCC compiler. Then we evaluate the performance gain from our SIMD DBT with helper function inlined and vector state mapping optimization in HQEMU-Vector with the GCC compiler. Finally, we analyze the composition of SSE instructions of SPEC 2006 CFP at static time and dynamic time to help explaining our results. The results of QEMU-Ori and QEMU-Vector are shown in Figure 20.

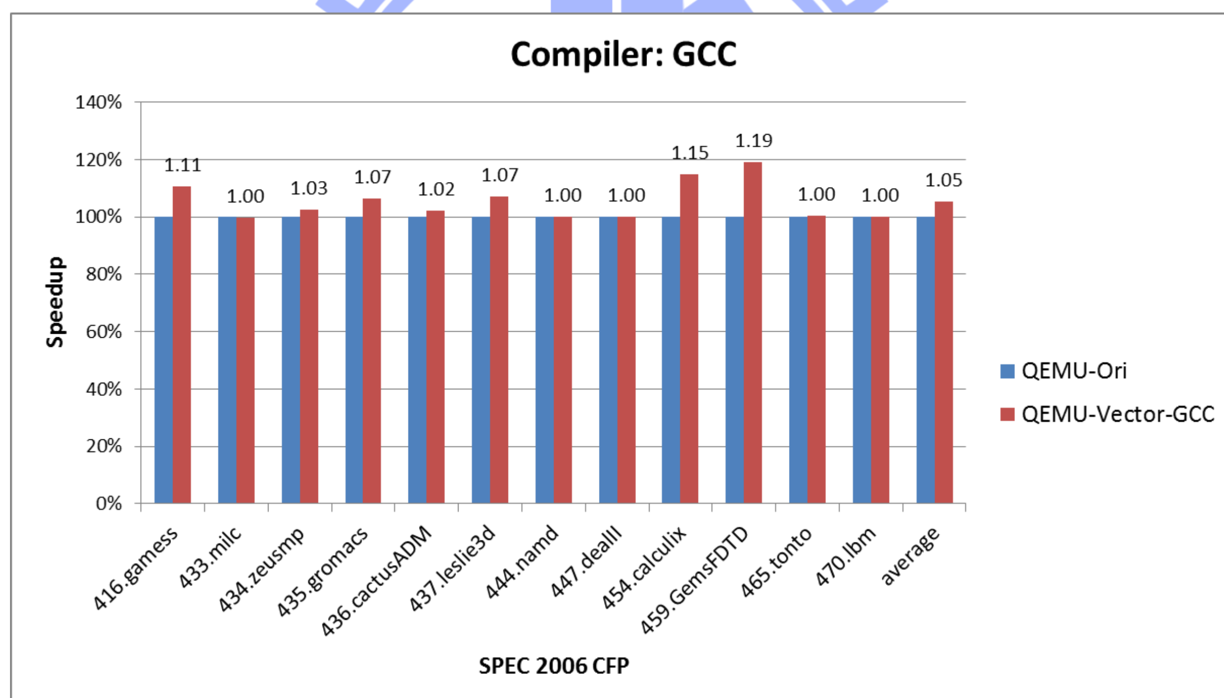


Figure 20. Speedup of QEMU-Vector compared to QEMU-Ori

In Figure 20, the execution time of QEMU-Vector has improved about 1.05X over QEMU-Ori, on average, for SPEC 2006 CFP. Several benchmarks (e.g. 433.milc, 444.named, 465.tonto, and so on) observed no performance gains. The overall improvement is also not impressive. This is due to two main reasons: the lack of helper function inlining

in QEMU is one major performance limiter and the proportion of real SSE instructions is not high in the SPEC binaries generated by the GCC compiler.

Then we evaluate the performance gain with helper function inlined and vector state mapping optimization in HQEMU-Vector. The results are shown in Figure 21.

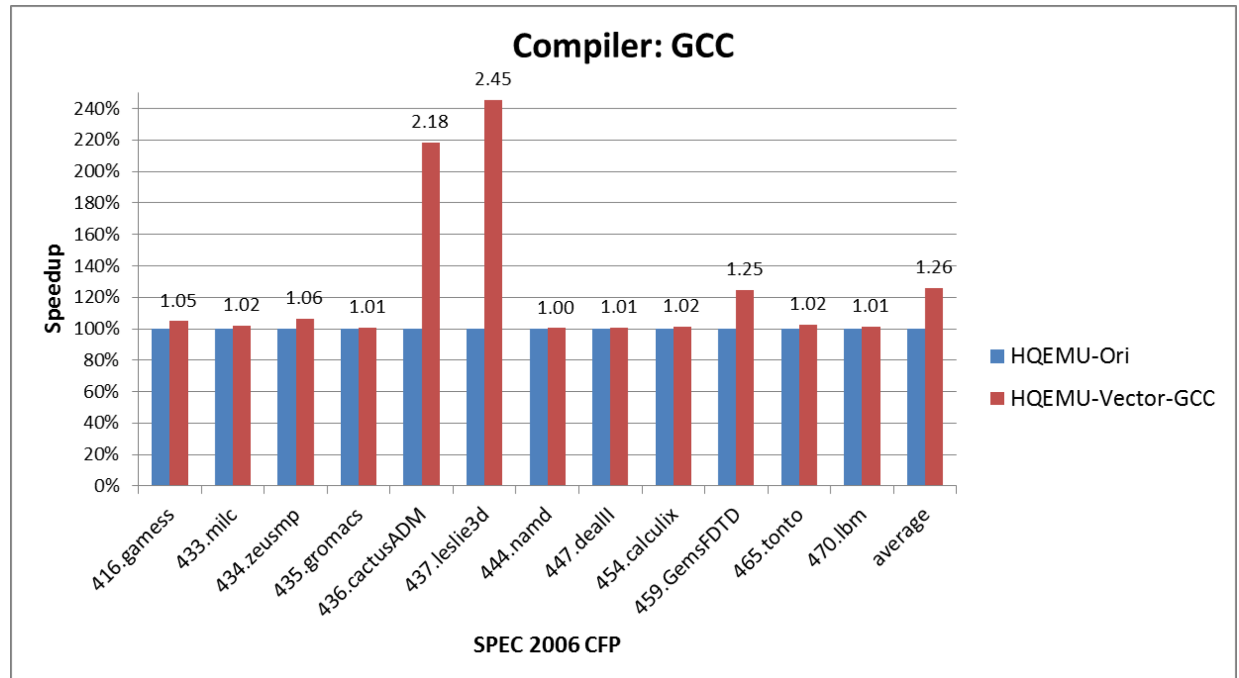


Figure 21. Speedup of HQEMU-Vector compared to HQEMU-Ori

From Figure 21, the execution time of HQEMU-Vector with helper function calls inlined and vector type state mapping optimization improves about 1.26X over HQEMU-Ori, on average, for the SPEC 2006 CFP benchmark. As shown in Figure 21, the improvement in 436.cactusADM and 437.leslie3d are very significant because these two benchmarks have a greater portion of instructions are real SSE instructions thus can benefit more from our SIMD code generation and optimization. Figure 22 and 23 show the proportion of SSE instruction at static time and dynamic time. From the results of Figure 20, 21, 22 and 23, we can observe that those benchmarks with low SSE proportions also have low speedups..

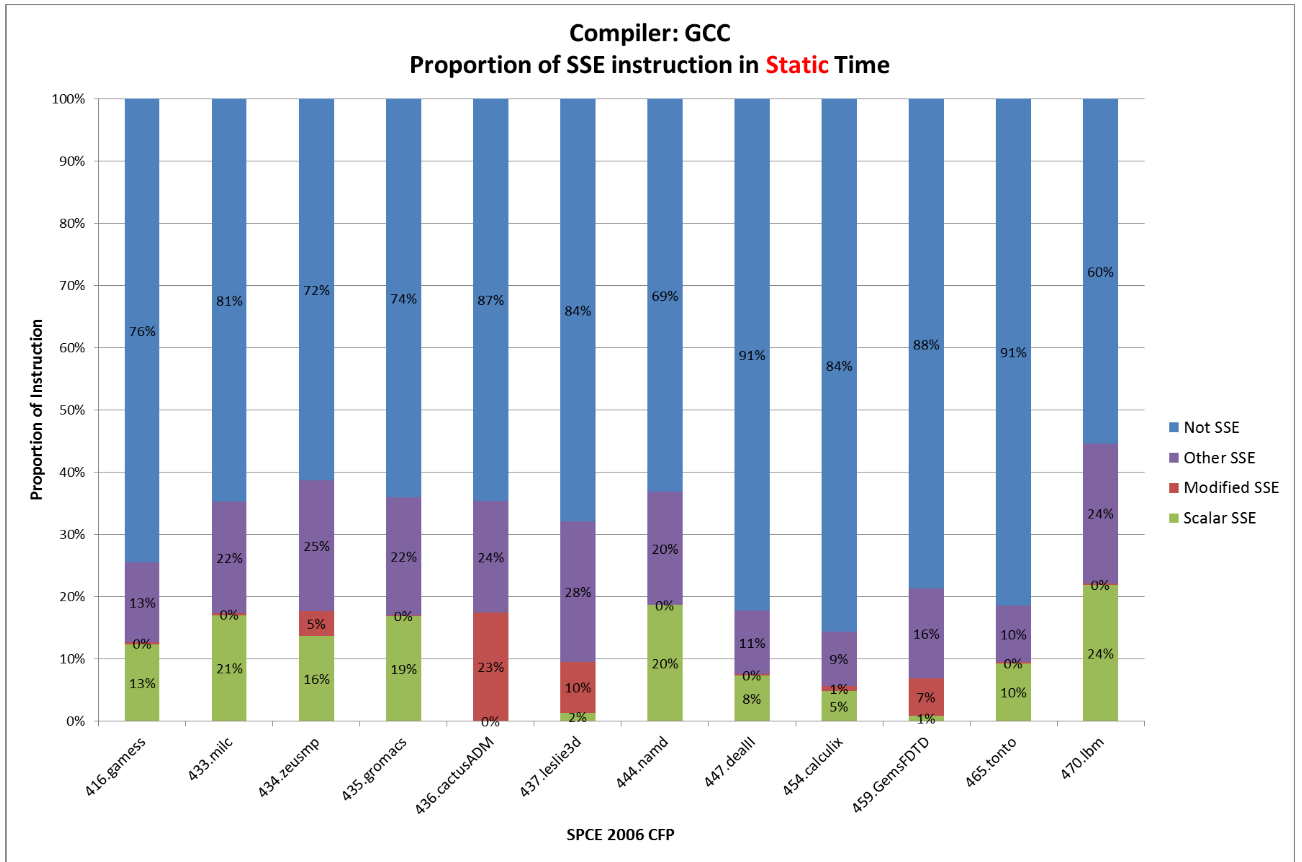


Figure 22. Proportion of SSE instruction at static time, using GCC

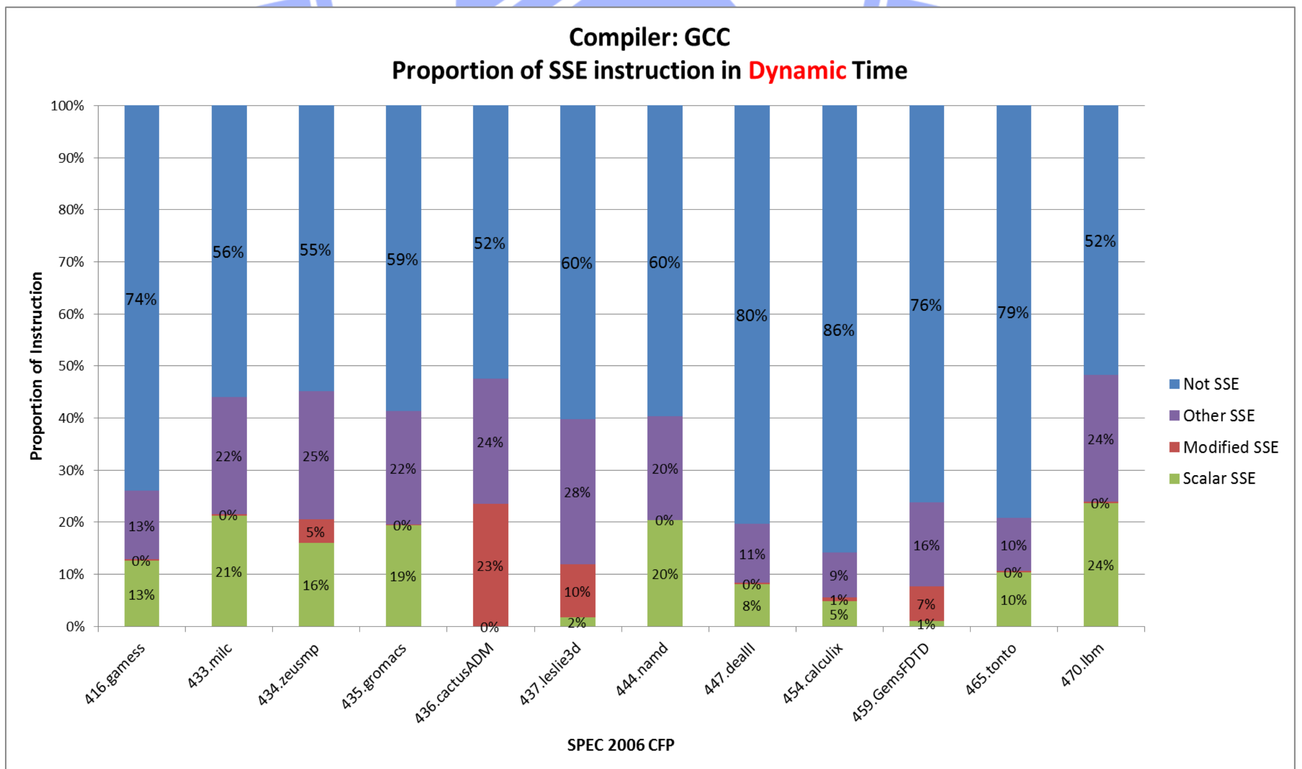


Figure 23. Proportion of SSE instruction at dynamic time, using GCC

5.2.2 Using the ICC Compiler

Different from the previous subsection, we use the ICC compiler instead of the GCC compiler for generating the guest binaries for testing our SIMD DBT approach. We want to compare the difference between using the GCC compiler and the ICC compiler, because it is well known that the ICC compiler would optimize for Intel architecture better and likely to generate more effective SSE instructions. The same sets of experiments in the previous subsection are conducted again using the ICC compiler generated binaries, and the results are shown in Figure 24 to 27.

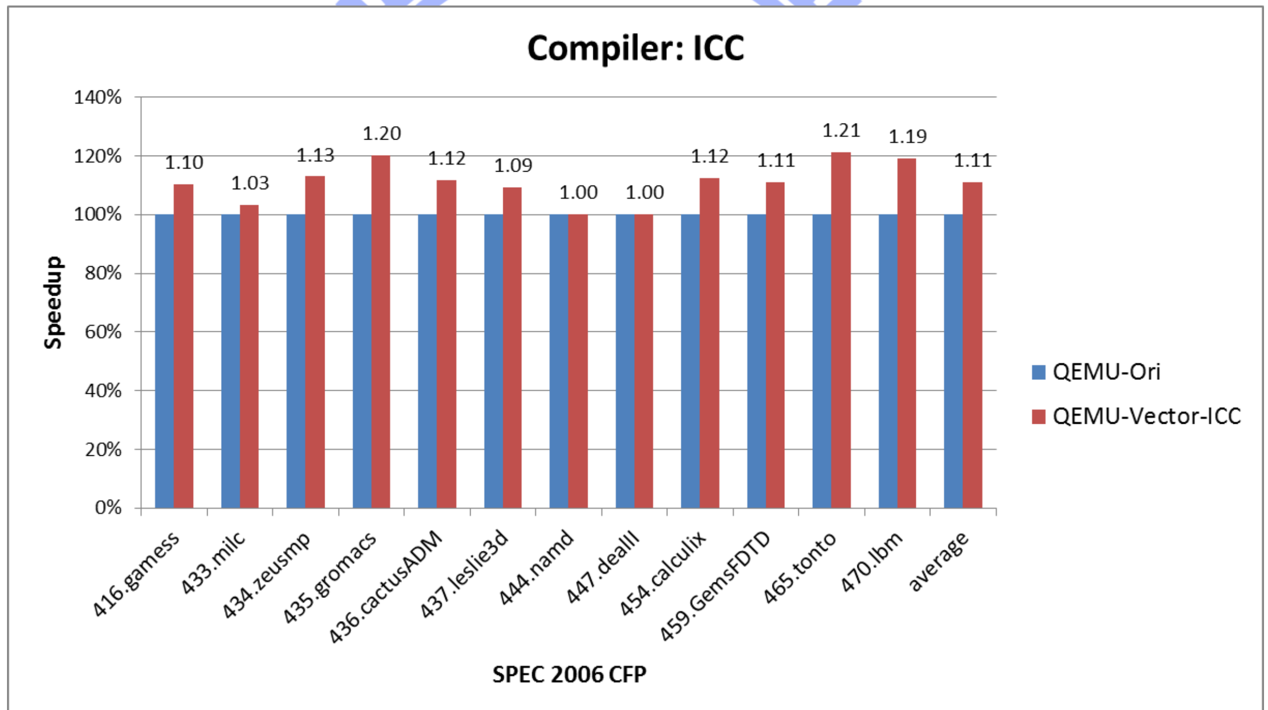


Figure 24. Speedup of QEMU-Vector compared to QEMU-Ori

From Figure 24, the execution time of QEMU-Vector improves about 1.11X over QEMU-Ori, on average. The speedup of from our SIMD DBT approach is greater here since the proportion of real SSE instructions in the benchmarks compiled by the ICC compiler is higher.

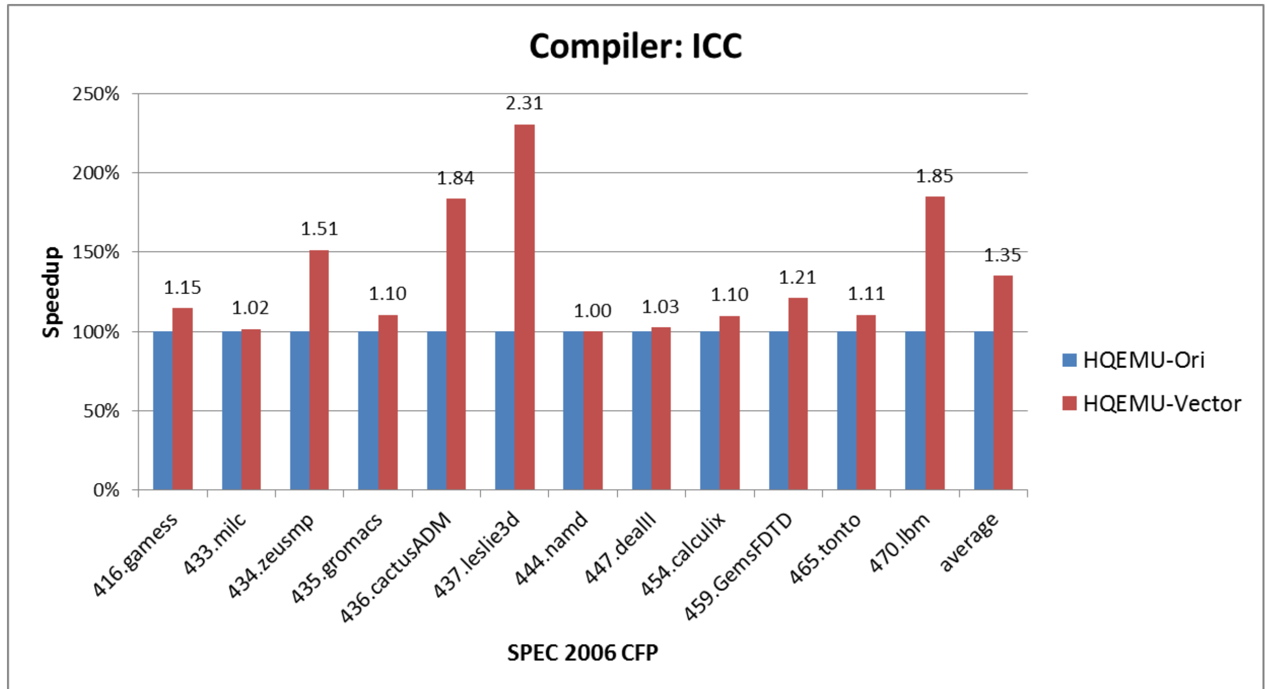


Figure 25. Speedup of HQEMU-Vector compared to HQEMU-Ori

From Figure 25, the execution time of HQEMU-Vector with vector type state mapping optimization improves about 1.35X over HQEMU-Ori, on average. The improvement is more impressive than the 1.26X speedup achieved for the binaries generated by the GCC compiler.

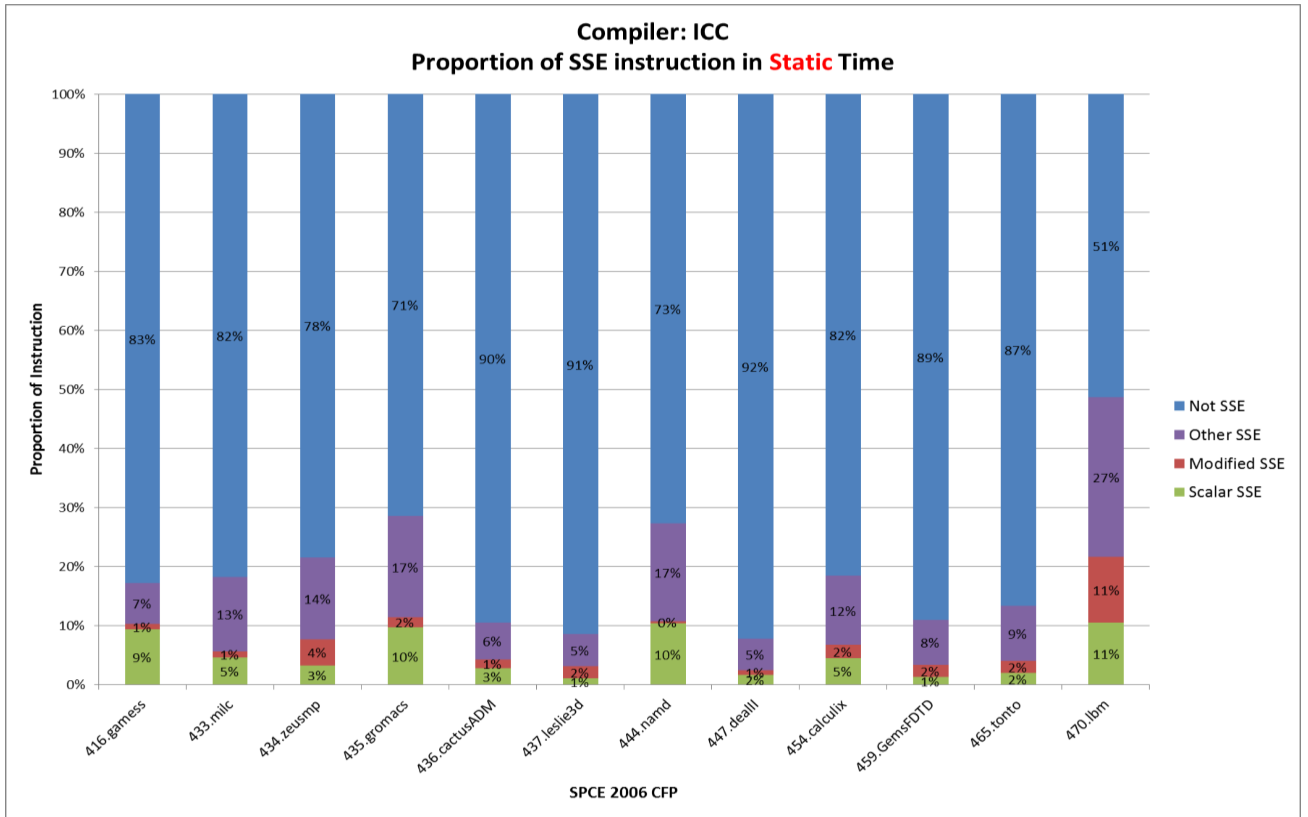


Figure 26. Proportion of SSE instruction at static time, using ICC

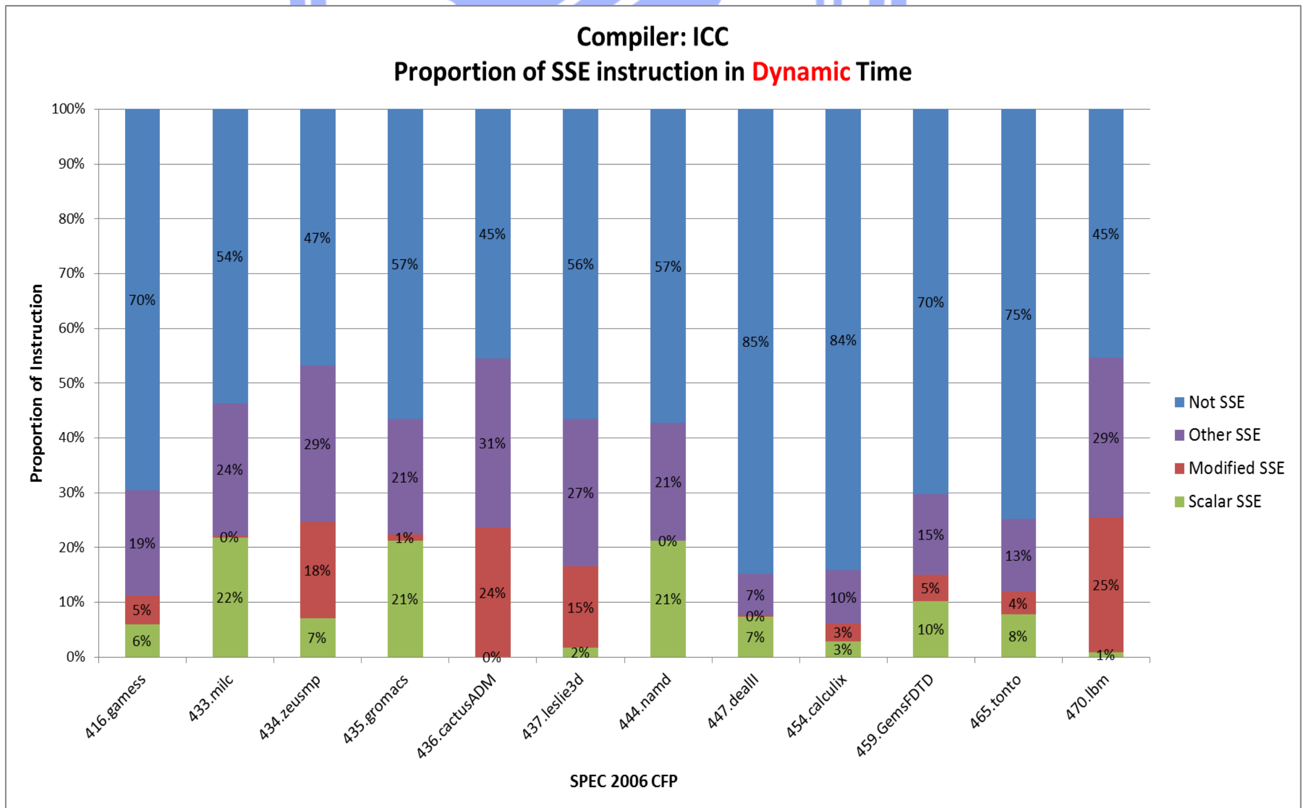


Figure 27. Proportion of SSE instruction at dynamic time, using ICC

Comparing Figure 22 and 23 to Figure 26 and 27, respectively, we can see that the proportion of SSE instructions is greater for ICC generated binaries than for GCC generated binaries. In particular, GCC generated binaries contain lots more Scalar SSE which do not benefit from our SIMD code generation approach. ICC is well-known for its "vectorization" capability. If a FP computation loop is vectorized, Vector SSE instructions will be generated, otherwise, Scalar SSE is generated instead. The higher portion of Vector SSE in ICC generated binaries shows that ICC can vectorize more effectively than GCC.

Note that there was a segmentation fault when emulating 410.bwaves, 453.povray, 481.wrf and 482.sphinx3 when compiled by Icc10.0 on HQEMU. 410.bwaves and 481.wrf also failed on the original QEMU, so we did not include these benchmarks in our benchmark set of the experiments in this section.

The next two figures show the speedup of QEMU-Vector compared to QEMU-Ori and HQEMU-Vector compared to HQEMU-Ori with GCC and ICC.

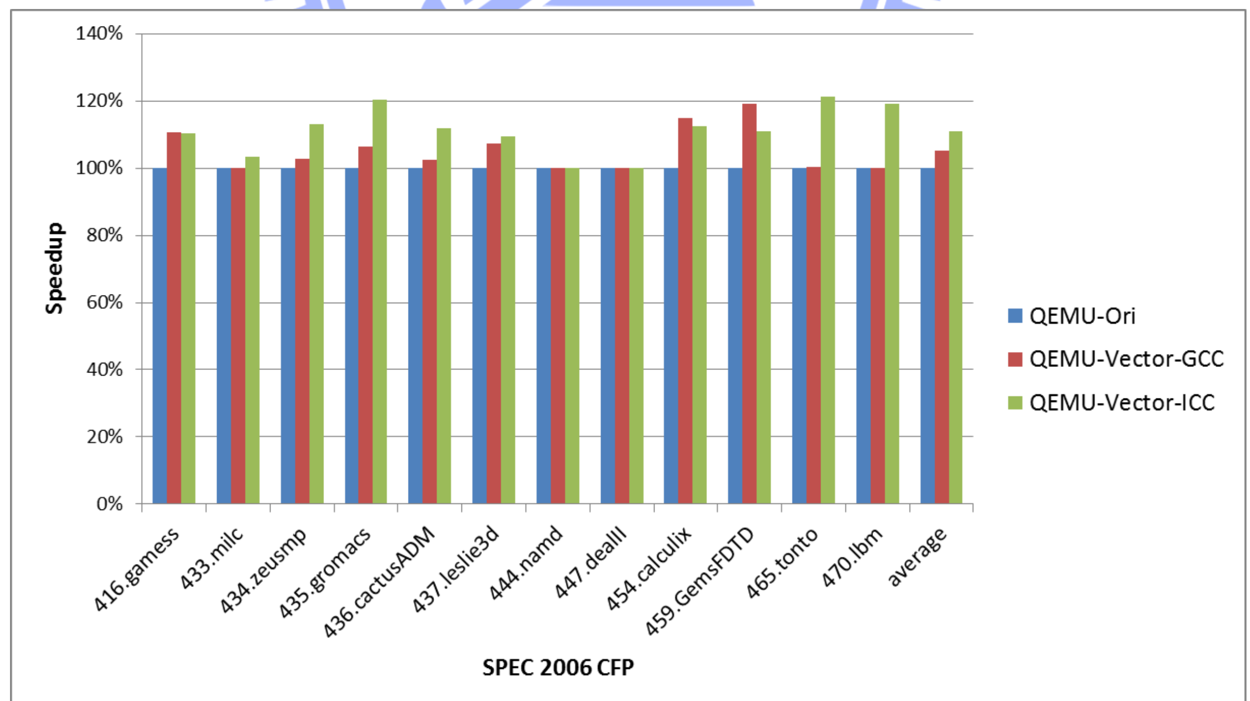


Figure 28. Speedup of QEMU-Vector compared to QEMU-Ori with GCC and ICC

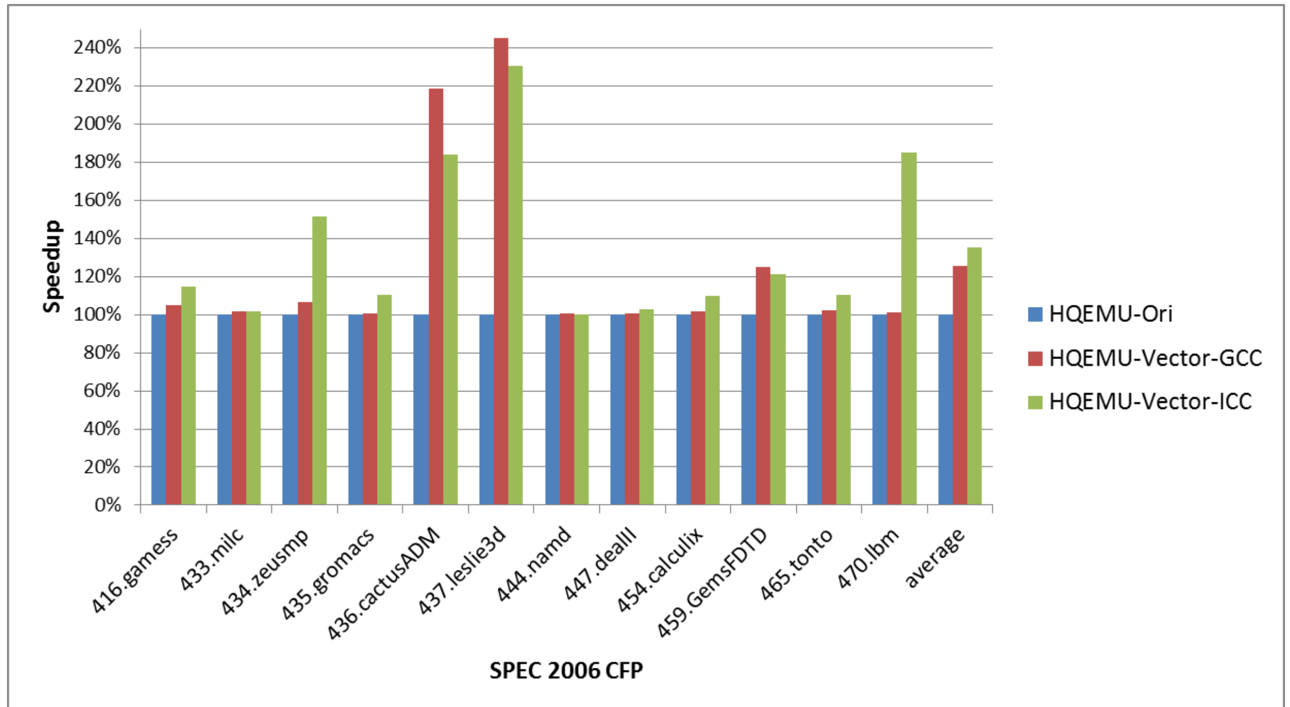


Figure 29. Speedup of HQEMU-Vector compared to HQEMU-Ori with GCC and ICC

From the results, the speed up of 470.lbm is not good for GCC generated code (1.01x speedup)but is great for ICC generated code (1.85x speedup). As we can see from Figure 30, nearly 50% of the SSE instructions generated by GCC are Scalar version instructions, thus cannot benefit from our SIMD code generation and optimization.

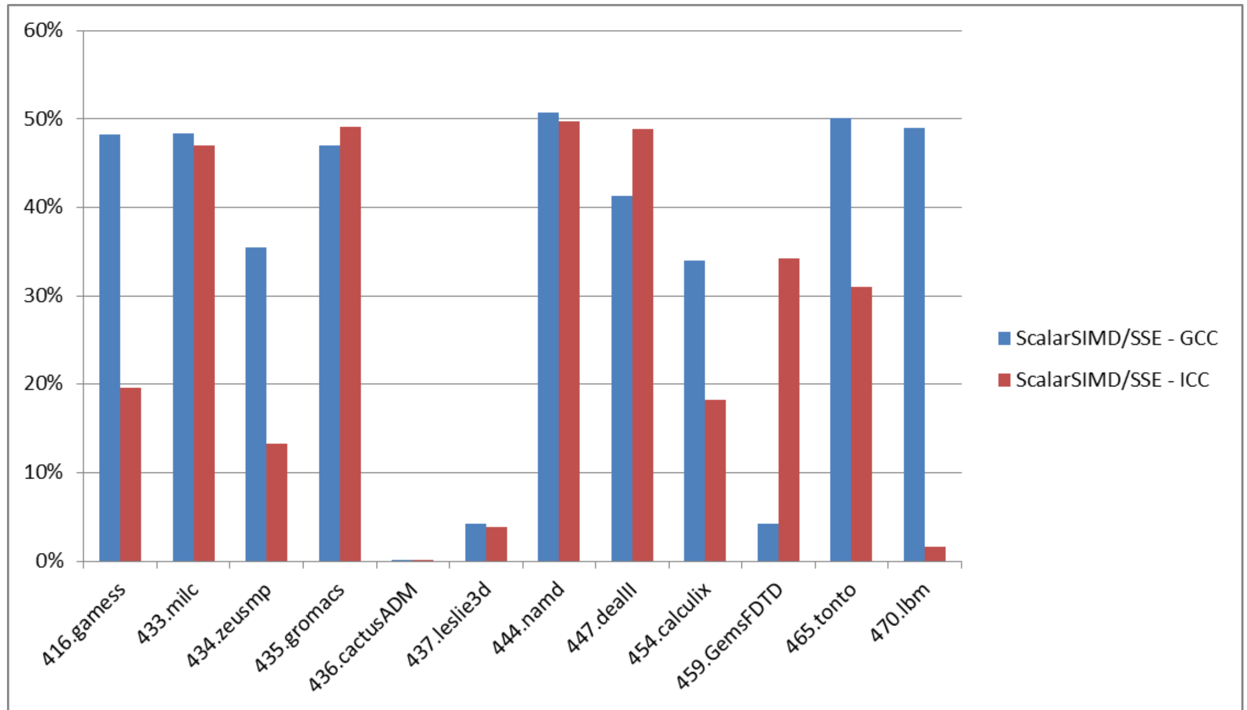


Figure 30. The scalar SIMD instructions ratio between using GCC and ICC

In Figure 30, we can see that the ratio of scalar SIMD instructions is from 10% to 50% except for 436.cactusADM and 437.leslie3d, which are lower than 10%. The lower the Scalar SSE ratio, the greater the speedup potential. This explains why 470.lbm compiled by ICC achieved 1.85X speedup, and both 436.cactusADM and 437.leslie3d gains more than 2X.

On average, using GCC generated code, the speedup from our SIMD DBT approach is 1.05X (as shown in QEMU-Vector) and 1.26X (as shown in HQEMU-Vector). The speedup for ICC generated binaries is 1.10X and 1.35X, in QEMU-Vector and HQEMU-Vector, respectively. This result indicates that the performance of our SIMD DBT approach will be heavily dependent on the proportion of Vector SSE instructions in the guest binaries.

VI. Conclusions and Future Work

In this thesis, we enhance the SIMD instruction generation capability in the DBT engine of QEMU/HQEMU to drastically improve their emulation efficiency for applications with SIMD operations. We using the Gcc vector extension which is powerful and portable, to replace the scalar instructions in the helper functions of SSE instructions to trigger host SIMD instructions to be generated on the host. Further inlining the helper functions can reduce function call overhead, as shown by the HQEMU implementation. In addition, we design and implemented a vector type state mapping optimization at the LLVM IR stage to increase the speed up from generated SIMD code. We have verified the implementation with the SPEC 2006 CFP benchmark suite, and we have conducted sanity check with simple loops on the performance achieved with our SIMD code generation method for the DBT engine of QEMU. Finally we use Gcc and Icc as compilers for SPEC 2006 CFP to test our improvement. The results of SPEC 2006 CFP show that the performance of HQEMU-Vector is 1.26X faster than HQEMU-Ori with the Gcc compiler, and 1.35X faster with the Icc compiler on average. The best case is 437.leslie3d where the speedup is 2.31X because it has more SSE instructions and benefits more from the vector state mapping optimization.

About future work, we have two directions. The first direction is to add new front-end, such as ARM NEON instruction generation of DBT engine. The other direction is to extend the built-in function of the Gcc vector extensions connected to HQEMU so that can generate more SSE instructions could be generated for x86 hosts machines.

References

[1] F. Bellard, “QEMU, a fast and portable dynamic translator,” in *Proceedings of the annual conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, 2005, pp. 41–41.

[2] D.-Y. Hong, C.-C. Hsu, P.-C. Yew, J.-J. Wu, W.-C. Hsu, P. Liu, C.-M. Wang, and Y.-C. Chung, “HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores,” in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, New York, NY, USA, 2012, pp. 104–113.

[3] C. Cifuentes and M. Van Emmerik, “UQBT: adaptable binary translation at low cost,” *Computer*, vol. 33, no. 3, pp. 60 –66, Mar. 2000.

[4] P. J. Drongowski, D. Hunter, M. Fayyazi, D. Kaeli, and J. Casmira, “Studying the performance of the FX!32 binary translation system,” in *In Proceedings of the First Workshop on Binary Translation*, 1999.

[5] D. Ung and C. Cifuentes, “Machine-adaptable dynamic binary translation,” *SIGPLAN Not.*, vol. 35, no. 7, pp. 41–51, Jan. 2000.

[6] K. Ebcioğlu and E. R. Altman, “DAISY: dynamic compilation for 100% architectural compatibility,” in *Proceedings of the 24th annual international symposium on Computer architecture*, New York, NY, USA, 1997, pp. 26–37.

[7] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, Washington, DC, USA, 2004, p. 75–.

[8]
L. Michel, N. Fournel, and F. Pétrot, “Speeding-up SIMD instructions dynamic binary tr anslation in embedded processor simulation,” in *Design, Automation Test in Europe Co nference Exhibition (DATE), 2011*, 2011, pp. 1 –4.

[9]
N. Nethercote, “Dynamic binary analysis and instrumentation,” A dissertation submitted for the degree of Doctor of Philosophy, University of Cambridge, November 2004.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991.

[11] "Low Level Virtual Machine(LLVM)," http://llvm.org

[12] “LLVM Language Reference Manual,” http://llvm.org/docs/LangRef.html

[13]
“Intel® 64 and IA-32 Architectures Optimization Reference Manual”, http://www.cs.pitt.edu/~kirk/cs3150spring2010/248966.pdf

[14] “Intel 80386 Reference Manual,” http://www.intel80386.com/simd/mmx2-doc.html

[15] “SPEC CFP 2006,” http://www.spec.org/cpu2006/CFP2006/

[16] “SSE introduction”, http://zh.wikipedia.org/zh-tw/SSE

[17] “QEMU”, http://qemu.org