國立臺灣大學電機資訊學院資訊工程學系
碩士論文
Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
Master Thesis

# 使用擴增暫存器增進 x86 處理器之效能

# Improving x86 Processor Performance via Extended Registers

吳哲仰
Che-Yang Wu

指導教授：徐慰中 博士
Advisor: Wei-Chung Hsu, Ph.D.

中華民國 105 年 7 月
July, 2016

doi:10.6342/NTU201603161

# 誌謝

感謝我的指導教授徐慰中老師，在研究以及論文寫作上給予寶貴的建議，讓我的研究可以順利的完成。感謝陳俊宇學長和傅勝余學長在研究計畫初期的貢獻，對我後續的研究幫助良多。最後感謝我的父母和雅喬在我研讀碩士學位期間的支持和鼓勵，讓我能專注在研究上。



doi:10.6342/NTU201603161

# 摘要

IA-32 為一個廣泛使用的指令集架構，為 x86 的 32 位元版本，指令的功能性很豐富但暫存器的數量則較少。如果在這個架構下我們可以擁有更多暫存器，藉由將更多變數保留在暫存器中或在指令排程上增加平行度，就有機會改進效能。雖然 64 位元版本的 Intel64 中暫存器的數量有所增加，但現存的 32 位元應用程式並沒有辦法利用到。其中不少嵌入式或桌面應用傾向於保持 32 位元避免額外增加的資料量。此外，作業系統或執行時期函式庫也可能需要重新編譯甚至重新開發。

在本論文中，我們設計了一個方法讓 32 位元的程式有機會利用到這些延伸暫存器。在我們的設計中，IA-32 的暫存器陣列被增加到 16 個，這個架構稱為 RegX16。32 位元的程式可以被編譯成 RegX16，但仍直接和舊有的 IA-32 函式庫連結，稱為混和模式 (mixed-mode) 二進制檔，同時包含了原架構和延伸架構的指令在其中。在這樣的架構中需要處理器模式 (processor mode) 來分辨當前指令的架構以正確解碼。執行時在遇到從 IA-32 的指令執行到 RegX16 指令時，處理器模式需要被切換。我們實作了一個編譯器可以利用到延伸暫存器，並在暫存器分配和指令排程時得到好處，並且自動地加入模式切換的指令。模式的切換是根據一個我們設計的方法，並且最佳化過以降低模式切換的負擔。

我們使用 EEMBC 效能測試集來評估 RegX16 對效能的改進，在純 RegX16 模式下最大的改進幅度為 19.5%，平均則有 10.9%。在測試程



doi:10.6342/NTU201603161

式和舊有的 32 位元函式庫連結的情況下，因為模式切換的額外負擔，平均的改進為 5.1%，其中非必要的模式切換已被我們以連結時期最佳化 (LTO) 削減，對某些程式下，混合模式仍然可以達到 21.2% 的加速。此外，我們也評估了延伸暫存器對指令排程的改善。我們精確的根據 RegX16 處理器的架構來設計我們的指令排程方式，以完全利用延伸暫存器帶來的優勢。在不使用延伸暫存器時，指令排程僅能帶來 3.9% 的加速，但一併使用延伸暫存器時，整體的效能改進了 9.7%。



doi:10.6342/NTU201603161

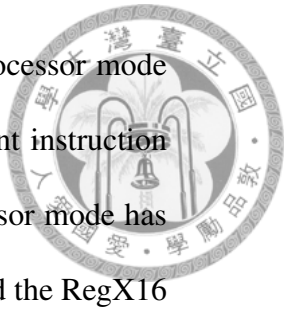
# Abstract

IA-32, the 32-bit version of x86, is a commonly used ISA (Instruction Set Architecture), which has feature-rich instruction set but only several registers. If there are more general purpose architectural registers defined in the ISA, the performance can be improved by promoting more variables to registers, holding more temporaries in registers, and exposing more ILP (Instruction Level Parallelism) for code scheduling. Although the 64-bit version, Intel64, has been extended with more registers, such extended registers cannot be exploited by 32-bit applications. Many embedded and desktop applications prefer to stay in 32-bit mode to avoid increased data working set. In addition, the operating system and runtime libraries have to be recompiled or even redeveloped for such a new architecture.

In this thesis, we design a mechanism which gives 32-bit applications an opportunity to exploit the extended registers. In our design, the general purpose register file in the original IA-32 is extended to 16 registers. We call this extended architecture RegX16. A 32-bit application could be recompiled to RegX16 yet still linked with the legacy IA-32 libraries (in executable format). Such an application binary is called mixed-mode binary, which is consist of



doi:10.6342/NTU201603161



instructions from both the original and the extended ISAs. Processor mode is introduced to identify which ISA is in use so that the current instruction can be correctly decoded. During binary execution, the processor mode has to be explicitly switched when transiting between the IA-32 and the RegX16 mode. We implement a compiler that automatically take advantages of the extended registers in both the register allocation and the code scheduling phases. Furthermore, our compiler also automatically inserts mode switching instructions to mixed-mode binaries according to our mode switching mechanism. Optimizations to reduce mode switching overhead are also in place.

The EEMBC benchmark suite is used to evaluate the performance improvement of RegX16, the greatest improvement observed is 19.5%, with an average speedup of 10.9 for the pure RegX16 binary. If the benchmarks have to be linked with legacy 32-bit libraries as mixed-mode binaries, the improvement is lowered to 5.1% on average due to the increased mode switching overhead. In the above experiments, we have exploited the link-time optimization (LTO) to eliminate unnecessary mode switching. For some applications, LTO has been quite effective, in one case, the mixed mode application still can get 21.2% of performance gain from the extended register. Furthermore, we also evaluate the performance improvement of exploiting extended registers on code scheduling. We have more accurately modeled the RegX16 micro-architecture to fully exploit the extended register in code scheduling. Our revised code scheduling model improves the performance by 3.9% without using the extended registers. When the extended registers are used, the average performance gain increased to 9.7%.



doi:10.6342/NTU201603161

# Contents





doi:10.6342/NTU201603161





doi:10.6342/NTU201603161

# List of Figures





doi:10.6342/NTU201603161

# List of Tables





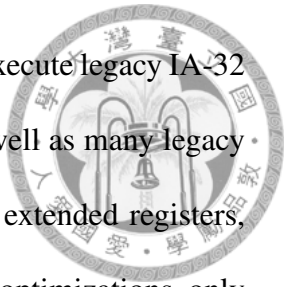
doi:10.6342/NTU201603161

# 1 Introduction

x86 [4] is a family of CISC (Complex Instruction Set Computing) instruction set architectures (ISAs) , which has variable length instructions, complex addressing mode, and feature-rich instructions but much fewer general purpose registers (GPRs) than modern RISC ISAs. IA-32, the first 32-bit version of x86 designed by Intel in 1985, has only eight GPRs. However, ARMv7 [7] and other earlier series already have 16 GPRs, and the latest ARMv8-A has 32 GPRs. During the evolution of x86, extensions to IA-32 have been continuously introduced. However, the number of registers in IA-32 has never been increased even if the processors actually have larger register file for renaming in out-of-order execution. In the 64-bit version of x86, called Intel64, the register file is extended to 16 GPRs to improve the performance of x86. Nevertheless, even though Intel64 processors can execute IA-32 code in compatible mode, the extended registers are not available for IA-32 applications.

Legacy applications have to be recompiled to exploit the new architectural features in a brand new ISA. Besides, redevelopment of operating systems and runtime libraries is required. In practice, to preserve investments of existing software, we often seek a mixed-and-match execution environment. For example, to exploit new architectural features, applications must be recompiled. However, such executables may want to link with the old mode libraries. This is because the source code of some old libraries may not be available for recompilation. How to work in such a mixed-and-match mode environment


doi:10.6342/NTU201603161



is highly desirable. RegX16 meets this goal in that the processor can execute legacy IA-32 binaries in IA32 mode, so we can run existing operating systems as well as many legacy applications. When new applications need to take advantage of the extended registers, we can also leverage existing compilers designed for x86 with mature optimizations, only retarget the register allocator and the code scheduler.

If we cannot recompile applications for some reasons, other techniques, such as Static or Dynamic Binary Translation (BT), can be used. BT is not the top choice for us due to the runtime overhead. Instead, we are interested in the mixed-and-match scenario where we can mix compiled binaries with legacy library binaries. We can compile the application programs to binaries in the extended ISA as long as we have the source codes, and then the compiled binaries and legacy library binaries are linked as a mixed-ISA application. For example, for an image processing application that uses a third-party library for loading images with multiple formats, we cannot reduce the image loading time, but we can improve performance of the image processing routines. A more general example is the C standard libraries, which are required by every program written in C.

It is relatively easy to add an instruction without breaking the compatibility if there are unused opcodes. For example, Streaming SIMD Extensions (SSE) do not conflict with the original instruction set in IA-32 and can be mixed in instruction granularity. Nevertheless, it is more difficult to extend the register file, since changing the number of registers will essentially impact all instruction encoding. In x86, the extended registers are only available in Intel64 and encoded with an extra byte, which is called REX prefix in the encoding method of Intel64. However, in other ISAs, we may not be able to extend the encoding method of instructions. For example, in RISC designs, since the instructions are fixed-length and may have exhausted all bits to encode themselves, there is no field remains for specifying extended registers. As an alternative, processor mode can be in-

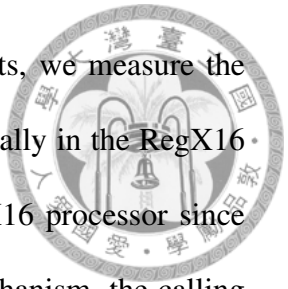

doi:10.6342/NTU201603161

troduced into such ISAs to identify how to decode and execute the instructions, and their encoding methods can be totally different from the original ISAs. Some modern ISAs already support mixed-mode execution, such as ARM. Two incompatible instruction sets, ARM and Thumb, can be mixed in the same binary, and ARM processors can automatically switch the processor mode according to mode bit, which is encoded as the least significant bit in the instruction addresses. For ARM, each instruction is at least 16 bits long, so the least significant bit is not used. For x86, this bit is not available since some instructions are single byte. Therefore, for RegX16, we need to handle mode switching by ourselves to ensure that each instruction is executed in correct mode. In this thesis, we design a software-based mode switching mechanism for mixed-mode binaries, which is fully independent to ISA features.

The case study of mixed-mode in this thesis is based on IA-32 and its extended ISA, RegX16, which is an experimental ISA designed by RDC [6] (RDC Semiconductor Co., Ltd.). RegX16 extends IA-32 by introduced the extended register file and REX prefix encoding of Intel64. Some IA-32 instructions are removed since they conflict with REX prefixes, but our compiler can avoid generating those removed instructions and substitute with other alternative instruction sequences. In the beginning, RegX16 processors are designed for RegX16-only bare-machines and machine control units (MCUs). No operating system or legacy libraries exist on such machines and therefore mixed-mode is not required. However, the processors can also be configured to enable RegX16 in user mode only to boot and run IA-32 operating systems. In such a system, RegX16 can be enabled individually for each process by itself. In addition, the processor mode, RegX16 or IA-32, can be passively switched during the execution of mixed-mode binaries.

Our implementation is based on LLVM [5], and leveraging the fairly robust and mature x86 backend. We also improve some optimizations which benefit from the extended


doi:10.6342/NTU201603161



register file and our mode switching mechanism. In the experiments, we measure the performance gain of code scheduling from extended registers, especially in the RegX16 processor we use. Static code scheduling is effective for our RegX16 processor since it is in-order. In addition, we found that in our mode switching mechanism, the calling convention can be changed incidentally with mode switching, and therefore we can use a customized calling convention in the RegX16 mode, which is intended to further reduce memory accesses.

The rest of the thesis is organized as follows. Section 2 provides the background of this work. Section 3 describes the design issues and the development of the compiler. Section 4 gives the benchmarks and our experiment set up, including results and analysis. Section 5 summaries and concludes the work.



doi:10.6342/NTU201603161

# 2 Background

In this chapter, we first introduce x86 ISAs, which includes the register file and how the registers are specified by instructions. Then we describe RegX16, an ISA extended from IA-32, and some problems of mixing RegX16 with IA-32. In the remains of this chapter, we introduce LLVM infrastructure, which is the base of our compiler for RegX16.

## 2.1 x86 Instruction Set Architectures

x86 is a family of ISAs, which includes 32-bit and 64-bit variants. The first 32-bit design of x86 architecture is IA-32, which used to represent 32-bit versions of x86 in this thesis. Besides, though the designs of 64-bit versions of x86 by Intel or AMD have some differences, they have same register files and most of the instructions. For convenience, we use Intel64 to represent all 64-bit versions.

### 2.1.1 Instruction Encoding

Both IA-32 and Intel64 are CISCs, which the length of instructions is variable. Each instruction is composed of several bit fields and encoded into several bytes. A bit field can be an opcode to specify the operation, a destination register index, or a source register index. Generally, there is no reserved bit in the encoding of an instruction [4].

For example, an `ADD r/m32, r32` instruction in IA-32 need two bytes. Figure 2.1



doi:10.6342/NTU201603161

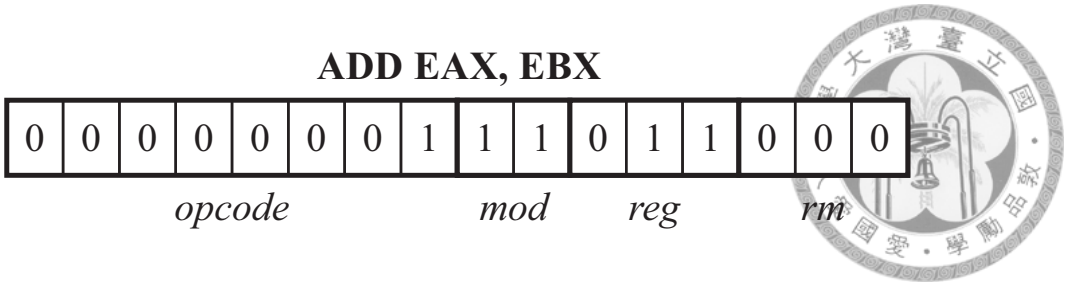



Figure 2.1: Opcode and ModR/M descriptor

shows an example that how an `ADD EAX, EBX` instruction is encoded. The first byte is the opcode and the second byte is a ModR/M descriptor. In this example, $opcode = 01\text{h}$ is for addition operations. In the ModR/M descriptor, the first two bits, field *mod*, specify the type of source operand; the next three bits, field *reg*, specify the source register, EBX; and the last three bits, field *rm*, are for the destination register, EAX. The index field has three bits, which is just enough for encoding the index of total eight registers. Since all bits are used to encode the instruction, there is no bit remaining for encoding more registers.

The *rm* field can also be used for indirect addressing, and sometimes an instruction may require a followed SIB descriptor for more complex addressing, which depends on *mod*. We will not introduce them in this thesis but briefly introduce the encoding of registers with ModR/M.

### 2.1.2 IA-32 Registers

There are eight GPRs can be used in most of the IA-32 instructions, and their data width are 32 bits. Naming of the registers follows the legacy convention below:

- EAX, ECX, EDX and EBX — accumulator, counter, data and base registers, used in general arithmetic operations. The indexes are 0–3, which are used in ModR/M for specifying registers. For example, in figure 2.1, the source, EBX, is encoded to $reg = 3$, and the destination, EAX, is encoded to $rm = 0$.
- ESP, EBP — stack pointer and stack base pointer, used to point out the top and the


doi:10.6342/NTU201603161

base of the stack. The indexes are 4–5.

- ESI, EDI — source index and destination index, used in some string operations on byte sequences. The indexes are 6–7.

Modern compilers, such as GCC and LLVM, do not follow the convention above during register allocation, except that ESP is reserved for the operations which are related to memory accesses of stack.

The least significant 16 bits of all GPRs in IA-32 can be accessed by their 16-bit versions, which are AX, CX, DX, BX, BP, SP, SI, and DI. The higher halves and lower halves of the first four 16-bit registers can be accessed by their sub-registers. The higher halves, the second significant byte of the 32-bit version, are identified by an 'H' suffix; the lower halves are identified by an 'L' suffix, respectively. AH, CH, DH, and BH are called high byte registers, and AL, CL, DL, and BL are called low byte registers.

The bit fields in an instruction only specify the register indexes of corresponding operands. The data widths of registers are decided by opcodes or prefixes of the instructions. In figure 2.2, for example, the opcode for adding two 32-bit registers is '01h'; adding 16-bit registers requires an additional operand-size override prefix, '66h'; adding 8-bit registers, has to use '00h' as the opcode.

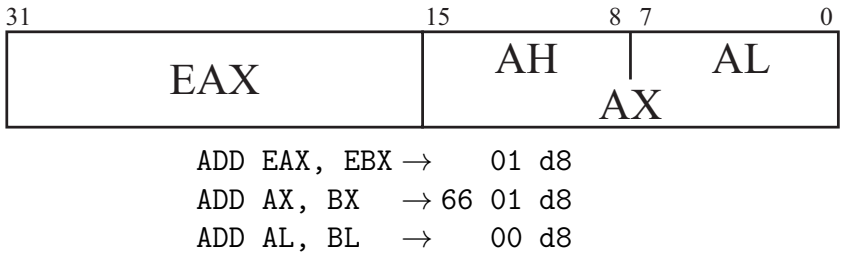
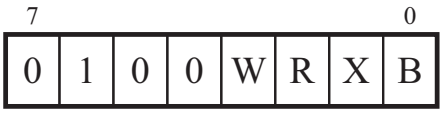



Figure 2.2: Register size and Encoding



doi:10.6342/NTU201603161




| Flag | Description |
|---|---|
| W | If set, the instruction uses 64-bit operand. |
| R | The extension bit of the *reg* field in ModR/M |
| X | The extension bit of the *index* field in SIB |
| B | The extension bit of the *rm* field in ModR/M |

Figure 2.3: REX prefix encoding

### 2.1.3 Intel64 Extended Registers

In Intel64, the register file is extended to 16 GPRs. All registers are extended to 64-bit, include the original ones in IA-32. The 64-bit versions of the original registers are RAX, RCX, RDX, RBX, RSP, RBP, RSI, and RDI. Besides, eight extended registers are named from R8 to R15, which also have the 32-bit versions with 'D' suffixes (R8D–R15D) and the 16-bit versions with 'W' suffixes (R8W–R15W).

Since no unused bit reserved for specifying extended registers in ModR/M descriptors, if any extended register is used, a REX byte is inserted as a prefix to the instruction. The encoding of REX prefix is showed in figure 2.3. When a flag bit is set in REX prefix, the corresponding operand is extended. For example, `ADD EAX, R11D` is encoded to '44 01 D8'. A REX byte, '44h', is inserted as the prefix. The destination index in field *rm* is set to 0 for EAX; the source index in field *reg* is set to 3, which specifies R11D but not EBX, since the R-bit is set. Figure 2.4 shows the register mapping with flag bit.

In Intel64, all registers have their 8-bit versions, which are AL, CL, DL, BL, SPL, BPL, SIL, DIL, and R8B–R15B. Not only the extended registers, but the last four original registers, SPL, BPL, SIL, and DIL, need to be encoded with REX prefix. The flag bit in REX and the field bits in ModR/M, totally four bits, are just enough to encode the index of 16 low byte registers. Therefore, the high byte registers, AH, CH, DH, and BH, can


doi:10.6342/NTU201603161

**32-bit Registers**

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| flag = 0 | EAX | ECX | EDX | EBX | ESP | EBP | ESI | EDI |
| flag = 1 | R8D | R9D | R10D | R11D | R12D | R13D | R14D | R15D |

**8-bit Registers**

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| without REX | AL | CL | DL | BL | AH | CH | DH | BH |
| flag = 0 | AL | CL | DL | BL | SPL | BPL | SIL | DIL |
| flag = 1 | R8B | R9B | R10B | R11B | R12B | R13B | R14B | R15B |

Figure 2.4: Register mapping with the index and the flag bit in REX prefix

be used only if the instruction has no REX prefix, and it is illegal to use both extended registers and high byte registers in an instruction at the same time.

## 2.2 RegX16 Instruction Set Architecture

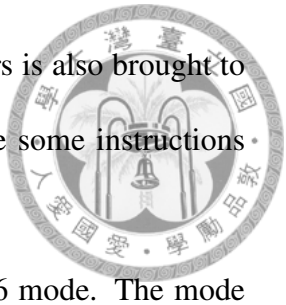
RegX16 is the ISA used in our case study, which is an experimental ISA designed by RDC (RDC Semiconductor Co., Ltd.) and currently implemented with FPGA board for development.

### 2.2.1 Design and Features

The instruction set of RegX16 is almost the same as IA-32, but the register file is extended to 16 GPRs, which exploit the design of Intel64 except the increased width. Therefore, REX prefix is also used to encode the extended registers in RegX16. All registers in RegX16 can be accessed in 32-bit, 16-bit, and 8-bit, with same register names and encoding method as Intel64.

The advantage to leverage the existing Intel64 design is to reduce the cost of hardware development, such as the register file and the instruction decoder. We can also leverage the implementation of Intel64 in existing compiler, especially the registers, assembly format,


doi:10.6342/NTU201603161



and instruction emitter. However, the restriction on high byte registers is also brought to RegX16. Even worse, not all IA-32 instructions are available, since some instructions conflict with REX prefix and have to be removed.

RegX16 processors can execute codes in IA-32 mode or RegX16 mode. The mode is controlled by the 28th bit in EFLAGS, which is reserved in IA-32. If the bit is set, the process mode is RegX16; otherwise, the processor mode is IA-32. We can use the instructions, PUSHF and POPF, to modify EFLAGS for mode switching. Since EFLAGS belongs to task contexts, the processor mode of each process is automatically preserved and resumed by the operating system when context switch occurs. Besides, to boot and run existing IA-32 operating systems, the processor automatically disables RegX16 on kernel codes by checking the Current Privilege Level (CPL) register. Because without modifying the operating system, we cannot catch the event that a process is switched to kernel mode and disable RegX16 before that.

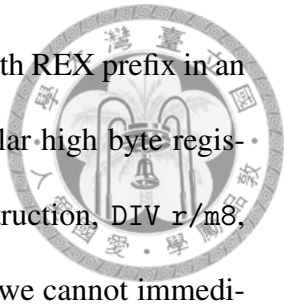
### 2.2.2 Conflicts of Instruction Encoding

REX prefix is used for encoding extended registers in RegX16. However, it conflicts with some IA-32 instructions and features.

Instructions `INC/DEC 32r` are removed from RegX16, since their opcodes overlap with REX prefix. These instructions increase or decrease 32-bit registers by one, of which the destinations are specified by 1-byte opcodes directly. The instructions require opcodes from '40h' to '4Fh' for increments and decrements of eight registers, but the range of REX prefix is from '40h' to '47h'. Though the opcodes of `DEC 32r` are disjoint to REX prefix, however, they share the same hardware with `INC 32r` and have to be removed together. If a legacy binary with a `INC/DEC 32r` instruction is executed in RegX16 mode, the processor will incorrectly decode the instruction as the REX prefix of the next instruction.



doi:10.6342/NTU201603161



Another restriction is that the high byte registers cannot be used with REX prefix in an instruction together. However, some IA-32 instructions need particular high byte registers for source or destination operands. For example, 8-bit division instruction, `DIV r/m8`, puts the remainder in AH and the quotient in AL. After the division, we cannot immediately use AH and extended registers in one operation. We have to move the result from AH to another compatible register, such as AL, if we want to use the result in an instruction with REX prefix.

### 2.2.3 RegX16 Processors

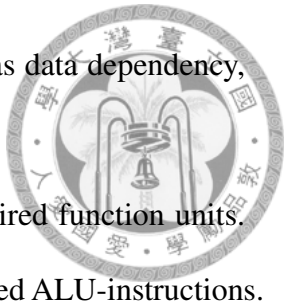
In this thesis, our designs and implementations are evaluated on the processor developed by RDC. Since RegX16 is an experimental architecture, the processor is not a real product but implemented with an FPGA board.

#### Instruction Issue

The processor is an in-order processor, but can issue at most three instructions in one cycle. Multiple instructions can be executed in the same time if the required resources, such as registers, do not overlap. For example, two addition instructions on different registers can be issued in the same cycle. Therefore, the extended registers also bring advantages to instruction level parallelism (ILP). A critical path can be broken into several shorter paths and be executed simultaneously, since the dependency of physical registers is broken.

There are some restrictions of instruction issuing on the RegX16 processor: first, the function units, two arithmetic logic units (ALU) and one address generation units (AGU) limit the maximum instruction can be issued because of structure hazards; second, if an instruction uses the AGU, then it must be the last one in the issue group; finally, two


doi:10.6342/NTU201603161



instructions cannot be issued in the same group, if two instruction has data dependency, which means read-after-write (RAW) on registers.

The instructions can be categorized into four classes by the required function units. General arithmetic instructions on registers require one ALU only, called ALU-instructions. Memory operations or addressing related instructions require the AGU, called AGU-instructions. If the instruction has to modify register and access memory, for example, a pop instruction that read the memory and increase the stack pointer register, requires one ALU and the AGU, called MIX-instructions. Finally, some complex instructions, such as division operations, have to degrade to several micro-instructions, which occupy all function units during execution and called uROM-instructions. In summary, maximal combination of instructions in an issuing group are: two ALU-instructions and one AGU-instruction, or one ALU-instruction and one MIX-instructions. The order is fixed since the AGU-instruction or MIX-instruction must be the last instruction in the group.

## 2.3 Mixed-mode Execution

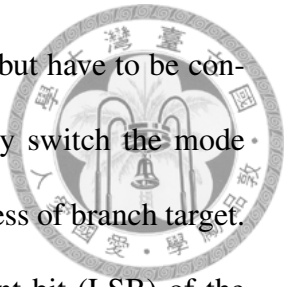
In this section we describe the mixed-mode execution and mode switching problems by two examples: ARM/Thumb and RegX16/IA-32.

#### Mixed ARM/Thumb

Thumb mode is introduced to ARM because of the code size issues. Thumb instruction set consists of 16-bit instructions, but the ARM instructions are 32-bit. Because of the smaller encoding space, Thumb instruction set has fewer operations and registers, but it is useful for compacting code of simple functions and less code size may result in better cache locality. Most modern compilers support generate mixed binary with ARM and Thumb instructions.



doi:10.6342/NTU201603161



ARM and Thumb instructions cannot be mixed in codes directly but have to be connected with branch instructions. ARM processors can automatically switch the mode when the branching happens because the mode is encoded in the address of branch target. The mode of targeting instructions is identified by the last significant bit (LSB) of the addresses, since ARM and Thumb codes are two-bytes aligned and the LSB is free to use. With such a hardware support, ARM and Thumb instructions can be even mixed in basic block granularity with low mode switching overhead.

**Mixed RegX16/IA-32**

Because the one-byte increments and decrements are removed, we cannot execute IA-32 codes in RegX16 mode. In the scenario we described, we can recompile our application to RegX16, but it still has to be directly linked with legacy libraries, which cannot be recompiled. Such a mixed-mode binary is composed of RegX16 and IA-32 codes, and they are mixed by function calls or jumps during execution.

During mixed-mode execution, we have to make processor mode be consistent with executing instructions. The RegX16 mode is controlled by the mode bit in EFLAGS, which requires several instructions to modify for either enabling or disabling. In the environment of our experiments, a mode switch requires about 14 cycles. It is much slower than a general instruction, which usually needs no more than two cycles.

To completely ensure the correctness, a naïve method is to wrap each RegX16 instructions with mode switches and every bundle now seems to be a single IA-32 instruction. However, the performance is extremely bad because of the overhead of frequent and redundant mode switches. Not only the correctness, but the performance should also be considered in designing mode switching mechanism.



doi:10.6342/NTU201603161

## 2.4 LLVM Infrastructure

LLVM is an open source collection of compilers and toolchains, which begins as a research project at the University of Illinois. LLVM is highly modular and support two-level compilation. High level language, such as C language, is first compiled to LLVM intermediate representation (IR) by the frontend; then the IR can be compiled to target binary code by the backend [2]. LLVM 3.8, which is the version used in this thesis, support over a dozen of backends, which are used to generate code for targeting ISAs.

Between frontends and backends, LLVM support a series of optimizations which do not depend on the target ISA. These optimizations that can be applied to LLVM-IR code, are called machine-independent optimizations. On the other hands, machine-dependent optimizations have to be applied in backends, since they require more information about the target. Besides, common optimizers are reused and shared among all backends.

Unlike optional optimizations, some works are necessary for code generation, for example, DAG lowering, instruction selection, register allocation, instruction encoding, etc. Each of these steps, include the optimizations, are handled by a particular "pass" in LLVM. We can customize LLVM by modifying existing passes or inserting new passes.

### 2.4.1 x86 Backend

The Intel64 and the IA-32 is handled in one backend. Since Intel64 is extended from IA-32, they very similar in many aspects. For example, most of the instructions have same encoding in both ISAs; The register definitions are similar, but Intel64 has eight extended registers and more low byte registers. Therefore, Intel64 and IA-32 are defined as "subtargets" of x86 in LLVM to make use of the common designs and implementations of the backend. For example, the instruction encoder is shared by both ISAs, except that



doi:10.6342/NTU201603161

it may insert REX prefix if targeting Intel64. There is also 16 GPRs in the definition of the register file by default, except that the extended registers are preserved in IA-32.

### 2.4.2 Register File and Calling Convention

The register file of an ISA has to be described in the backend. The names, number, and width of registers have to be defined in the TableGen description file. In another description file, the calling convention is defined, which includes the callee-saved and caller-saved registers and the how arguments are passed in function calls.

In register allocation, though the algorithm is generalized, the use of registers can be partially controlled by the backend. In x86, for example, there are 15 registers can be used in register allocation, but a getReservedRegs method is used to specifying the forbidden registers in IA-32. Besides, the classes of registers are also defined, which are used to define the format of instructions. For example, a `MOVZX r32, r/m8` instruction is defined to use a GR32 register for the destination and a GR8 register for the source in Intel64, but the source register can also be GR8_ABCD_H, which is the class of high byte registers, in IA-32.

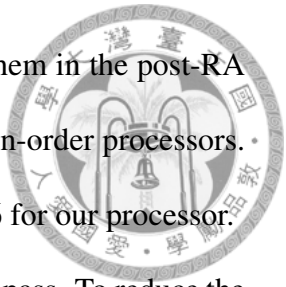
Unlike the registers, the calling convention is defined individually for each subtarget. Only the callee-save registers (CSRs) have to be defined, and LLVM assume that other GPRs are caller-save. In addition, how arguments are passed is defined by a sequence of operations, which include type checking and corresponding actions for arguments.

### 2.4.3 Instruction Scheduling

In LLVM, instruction scheduling is separated to two phases: the pre-RA scheduling and the post-RA scheduling. The pre-RA pass is executed before register allocation, which aims to balance register pressure and ILP. After register allocation, some instructions for



doi:10.6342/NTU201603161



spilling and restoring register are inserted, and we have to schedule them in the post-RA pass. Post-RA scheduling helps minimizing the stalls in pipeline for in-order processors. In our implementation, we extend the post-RA scheduling pass in x86 for our processor.

The instructions are scheduled in basic block scope in the post-RA pass. To reduce the stalls, the pipeline structure is required, which is defined by the itineraries of instruction classes. An itinerary describe how the instruction will be pipelined by the setting of the timing, required function units, and latency of each pipeline stage. Besides, an additional hazard recognizer can be introduced into the scheduler to customize hazard detection.

The post-RA scheduler uses a greedy algorithm for instruction scheduling, which is known as list scheduling. Before scheduling a basic block, the dependency is analyzed and the DAG of instructions in the basic block is created. The scheduler repeatedly selects an instruction in the DAG, which does not depend on other instructions; emit it to the scheduled code; and remove it from the DAG. In practice, the available instructions are placed in a priority queue, sorted by some heuristics with the height and out-degree of the node, and selected in order.



doi:10.6342/NTU201603161

# 3 Design and Implementation

## 3.1 RegX16 Code Generation

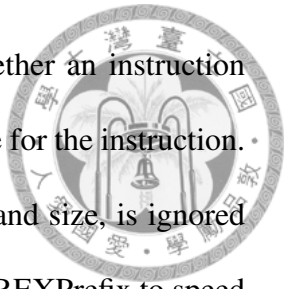
We extend the x86 backend in LLVM to support RegX16 architecture. Unlike IA-32 and Intel64, RegX16 is not treated as a subtarget but an extension of IA-32 in our implementation, and therefore we can reuse the source code of IA-32. However, since REX prefix conflicts to the one-byte increments and decrements and high byte registers, merely extending the registers and the encoding method is not enough. In addition, we have to disable these conflicting features in RegX16.

### 3.1.1 Extending Register File

The extended registers can be used in RegX16 by inserting REX byte as prefix of the instruction. We do not need to define these registers again, but can exploit the definition of Intel64 registers. Since we do not reserve the extended registers in RegX16, they can be assigned by the register allocator, which include the extended low byte registers, SPL, BPL, SIL, and DIL.

In IA-32, the code emitter does not correctly encode the extended registers. Since registers are encoded with the last three bits of their indexes, the extended registers are incorrectly encoded to corresponding original registers. We port the DetermineREXPrefix


doi:10.6342/NTU201603161



function to IA-32, which is originally used in Intel64 to check whether an instruction needs REX prefix. The function also calculates the required REX byte for the instruction. However, since the W-bit in REX, which is used to specify the operand size, is ignored by RegX16 processors, we can remove the related code in DetermineREXPrefix to speed up compilation.

### 3.1.2 One-byte Increments and Decrements

`INC r32` and `DEC r32` instructions cannot be used in RegX16, since the opcodes overlap to REX prefix. Fortunately, we can use `INC r/m32` and `DEC r/m32` instructions, which support operations on both register and memory. These instructions need two bytes, which is composed of the opcode 'FFh' and the destination descriptor *rm*.

The one-byte increments and decrements are selected in the instruction lowering pass by X86MCInstLower. The instruction lowering pass is used to translate and simplify instructions. In IA-32, an `INC r/m32` which generated from instruction selection, will be convert to `INC r32` in the instruction lowering pass. In our implementation, we disable such conversions in X86MCInstLower when targeting RegX16.

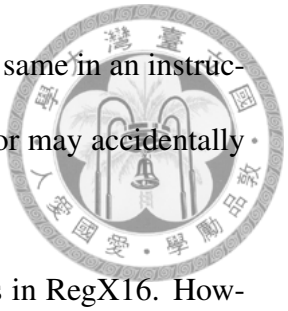
### 3.1.3 High Byte Registers

All high byte registers, AH, CH, DH, and BH, cannot be used in an instruction with REX prefix because the indexes of these registers are occupied by BPL, SPL, DIL, and SIL.

In Intel64, high byte registers are reserved, even though they can be used in the instruction without REX prefix, and the number of available byte registers is reduced from 12 to 8. This is because of the limitation of generalized register allocators in LLVM, which cannot prevent to use high byte registers with REX prefix. Currently, the register type can be individually set for each operand by specifying register classes, such as GR32



doi:10.6342/NTU201603161



or GR8. However, we cannot force the types of all operands to be the same in an instruction. Therefore, if high byte registers are allowed, the register allocator may accidentally select an illegal combination of registers.

For the same reason, we have to prevent using high byte registers in RegX16. However, even if high byte registers are not used in register allocation, they can be directly selected by some optimization passes in IA-32. For example, a peephole pattern in IA-32 is used to recognize a right-shift of byte by eight bits. Such an instruction is replaced by a copying to GR32_ABCD and a copying from their high byte halves, since they have opportunity to be eliminated by copy propagation or coalescing with other copying.

Such optimizations can be fully removed to prevent using high byte register accidentally, but we use a different approach, which also attempts to retain the optimizations. The approach needs an additional pass after such optimizations have been done. The pass scans the instructions and check if any source may be high byte register. If so, the destination is forced to select original registers, which makes no REX prefix is required in the instruction. In the worst case, one additional copying is required for moving data from the original register to the other extended register, but we still do not use more instructions than the non-optimized code.

### 3.1.4 Granularity of Mixed-mode

Our implementation support mixing RegX16 and IA-32 code in a binary. The granularity affects the design of mode switching, which is described in the mixed-mode section. Here we merely describe how we support mixed-modes of different granularities.

Assume that all functions in a module is compiled to RegX16, but the module has to be linked with other modules which the mode is unknown. For example, we have the source codes of our applications but the runtime library is pre-compiled. A processor


doi:10.6342/NTU201603161

feature flag, HasRegX16, is added to our implementation, and we can enable RegX16 for the whole module by compiler flags.

If not all the functions are going to be compiled to RegX16, we have to support mode controlling in function granularity. In our implementation, we add a new function attribute, __regx16, to the compiler framework. The attribute has to be support by both frontend and backend, since it has to be recognizable in high level language. We can append this attribute to the functions which have to be compiled to RegX16, and we can also use it to identity external RegX16 functions by appending it to their declarations. Besides, in our link-time optimization (LTO) support, the attribute is automatically appended to RegX16 functions.

In summary, RegX16 can be enabled in either module or function granularities. When RegX16 is enabled for whole the module, the compiler conservatively assumes that the modes of external functions are unknown. However, if we specify the mode of each function, the compiler has more information that can be used to optimize mode switching.

Our implementation does not support smaller granularities than a function, such as a basic block or an instruction, since register allocation is applied on whole function, and there is no reason to limit extended registers to be used in some basic blocks only.

## 3.2 Mode Switching

When mixed-mode is used, our implementation automatically generate mode switches to ensure that every instruction is executed in correct mode.



doi:10.6342/NTU201603161

### 3.2.1 Mode Switching Overview

Since the mode is controlled by the 28th-bit of EFLAGS, which can be modified by several instructions, no extra instruction for mode switching is supported by RegX16 processor. The mode can be switched by following instructions:

| **Switch to RegX16** | **Switch to IA32** |
|---|---|
| `PUSHF` | `PUSHF` |
| `OR DWORD [ESP], 0x10000000` | `AND DWORD [ESP], 0xEFFFFFFF` |
| `POPF` | `POPF` |

The first instruction, `PUSHF`, write EFLAGS to stack, since there is no instruction for setting ELFAGS directly. Then, we use `OR 0x10000000` or `AND 0xEFFFFFFF` to modify the value for enabling or disabling RegX16. Finally, the value is write back to EFLAGS by `POPF`. For convenience, we use *switch_to_regx16* and *switch_to_ia32* for the shorten of the above instructions.

A naïve mechanism of mode switching is to wrap each RegX16 instruction by the mode switches, *switch_to_regx16* and *switch_to_ia32*, and the instruction seems to be IA-32. However, this mechanism is impractical because of the overhead of mode switching. Since the pipeline has to be flushed when mode changes, the instructions of mode switching cannot be pipelined and takes 14 cycles in our processor. Therefore, enabling and disabling RegX16 for an instruction requires extra 28 cycles, which is not acceptable.

A better mechanism is to switch the mode for functions, not instructions. We have to insert mode switching at the boundary of function. More specifically, when a function is called or returns, we have to switch the mode according to the targeting function. The mode can be switched before branches happen. For example, assume that there is a call from RegX16 function to IA-32 function. We can add *switch_to_ia32* before the call instruction. Otherwise, the mode has to be switch at the target of branches. In the above example, if no *switch_to_regx16* is inserted before the return instructions of the IA-32



doi:10.6342/NTU201603161

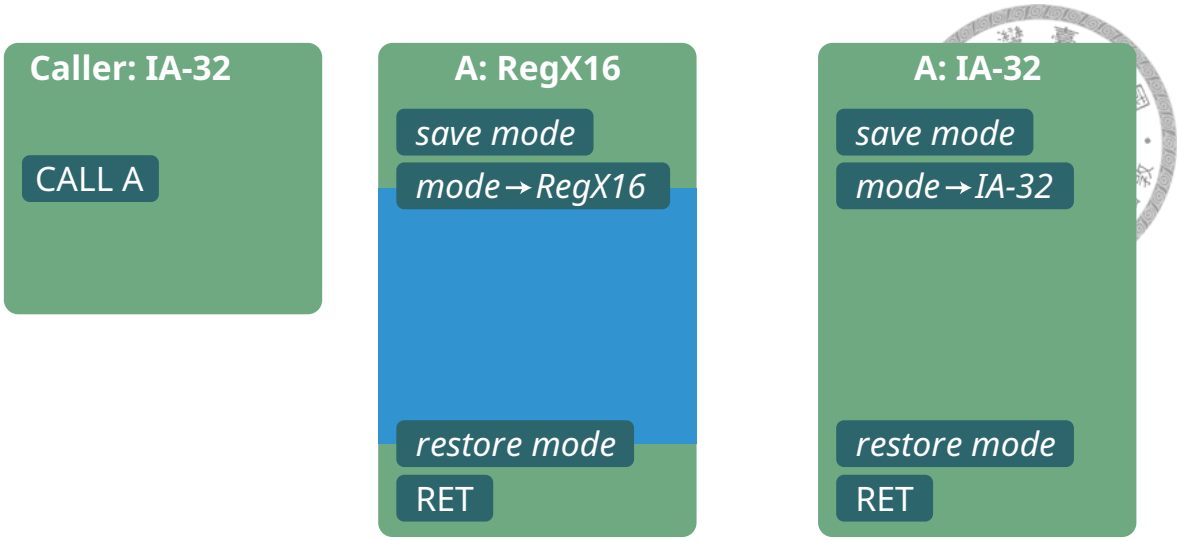



Figure 3.1: Callee-switch mode switching mechanism

function, we have to insert *switch_to_regx16* after the call instruction. We first introduce two basic mechanisms, callee-switch and caller-switch, in following paragraphs.

**Callee-switch**

In callee-switch mechanism, the callee is responsible for mode switching. At the entry point of each function, *switch_to_regx16* is inserted if it is a RegX16 function, and *switch_to_ia32* is inserted for IA-32 function. However, at the exit points, we cannot determine which mode should switch to, since the return instructions are indirect branches, which the targets are unknown or variable. Therefore, we have to save the mode of the caller by push EFLAGS onto the stack, which is similar to reserve callee-saved registers. Then we can restore the mode for caller at exit points. In figure 3.1, function A is called from IA-32 caller. If function A is a RegX16 function, the mode is switched to RegX16 before entering the function body and restored to IA-32 before leaving the function. If function A is IA-32, the mode is switched to IA-32, which is actually not changed, and retains IA-32 when leaving the function.

The disadvantage is that when the caller and the callee have the same mode, mode switches are redundant. Furthermore, the callee-switch mechanism is failed when there


doi:10.6342/NTU201603161

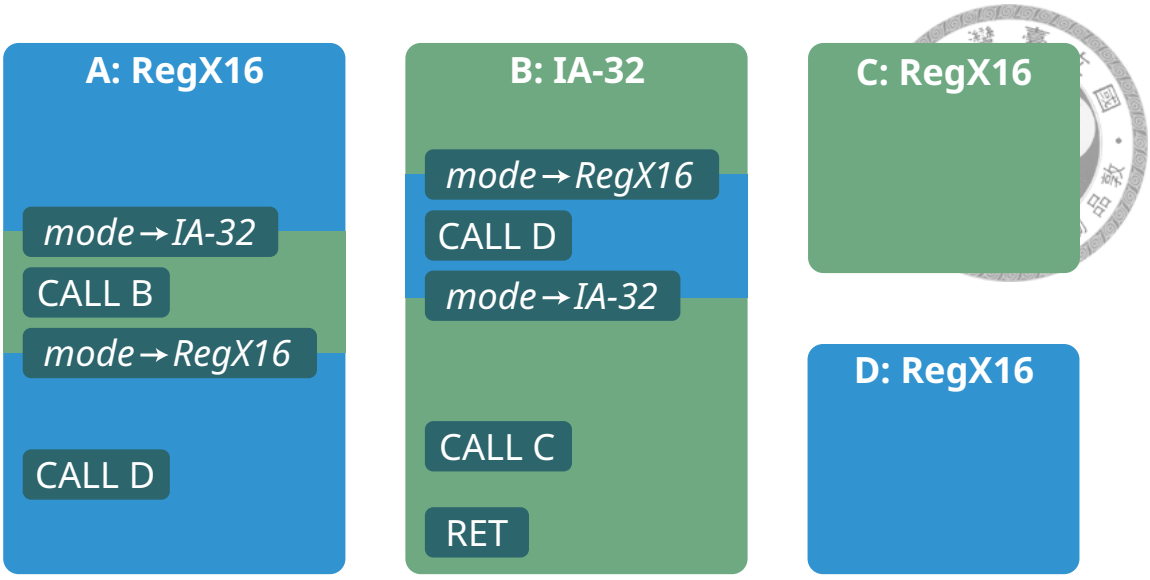



Figure 3.2: Caller-switch mode switching mechanism

are external calls to pre-compiled functions, since we cannot insert mode switches in such callees.

**Caller-switch**

In caller-switch mechanism, the caller is responsible for mode switching, and mode switchings are inserted before and after call instructions. Before a call, the mode should be switch to be the mode of the callee, and it should be switch back to the mode of the caller. We do not have to reserve EFLAGS, since the mode of callee and caller is determinate, and therefore, caller-switch is a little bit faster than callee-switch. Moreover, when the caller and the callee are in the same mode, the mode switching can be removed to prevent redundancy. In figure 3.2, function A is RegX16 and B is IA-32, the mode is switched at every function calls between different ISAs. If the caller and callee have same ISA, such as the call from function B to function C, we do not have to insert mode switches.

Caller-switch does not support indirect function calls since modes of the callees are unknown. To solve this problem, a function pointer is limited to point to functions with


doi:10.6342/NTU201603161

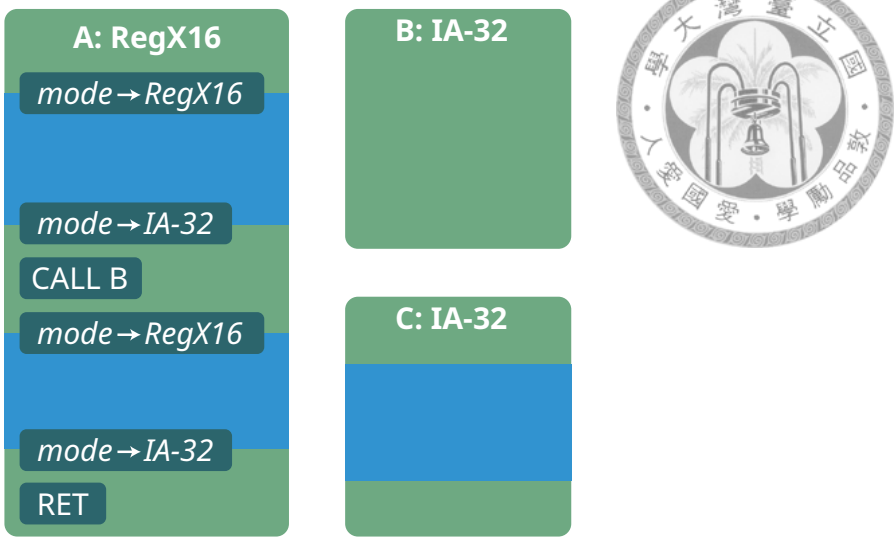



Figure 3.3: Legacy compatible mode switching mechanism

single mode. In our compiler implementation, a function pointer can be limited to RegX16 functions by the appending __regx16 attribute to the declaration of the pointer.

### 3.2.2 Legacy Compatible Mode Switching

The mode switching mechanism used in our implementation is fully compatible with linked libraries. Our design is a combination of callee-switch and caller-switch and optimized to eliminate redundant mode switching.

The idea is to make every function seem like IA-32. No IA-32 function has to be modified, and all mode switches are inserted into RegX16 functions. Therefore, the functions in legacy libraries do not have to be recompiled. Both callee-switch and caller-switch are required for RegX16 functions. Callee-switch makes the functions seem like IA-32 and able to be called directly from other functions. According to caller-switch mechanism, switches are inserted at each function calls in RegX16 functions, since all called functions seem like IA-32. With this mechanism, RegX16 functions are able to call external IA-32 functions and safe to be passed as function pointers. In figure 3.3, function A is compiled to RegX16 with mode switches at the entry, exit, and call site.


doi:10.6342/NTU201603161

In the following paragraphs, we describe the optimizations which can improve the performance of our mode switching mechanism. The following optimizations are only applied to RegX16 functions, since we assume that we cannot modified pre-compiled IA-32 functions.

### Optimize Direct Calls

The disadvantage is that redundant mode switches exist between two RegX16 functions. Ideally, if both the caller and the callee are RegX16 functions, no mode switch is required, but we switch twice in this case. Figure 3.4 shows the problem. When function A calls function B, the mode is switched to IA-32 before the call and switched back to RegX16 after entering the callee function. Same problems occur when the callee functions return.

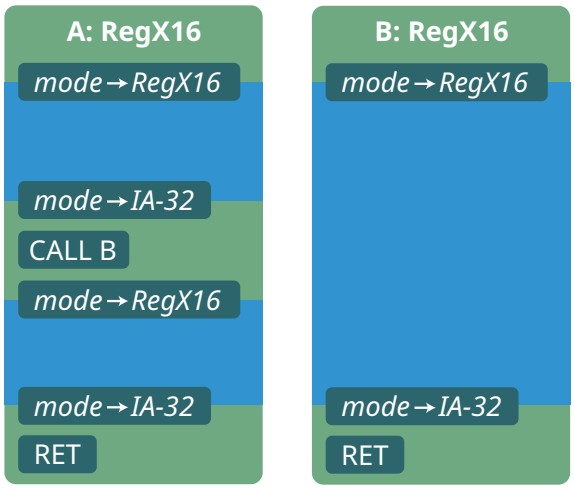



Figure 3.4: Redundant mode switches in a RegX16 to RegX16 call

The redundant mode switch in the caller can be removed directly, but the one in the callee cannot be removed, since the callee function may be called by other IA-32 functions without mode switching.

To remove the redundant mode switch in the callee, we have to migrate the switch to a wrapper function. In our implementation, the name of the wrapper function is same as the


doi:10.6342/NTU201603161

callee, and the original callee function is renamed by appending a ".regx16" suffix. The wrapper function switches the mode and call the original callee directly. All uses of the original callee is replaced by the wrapper function, but except for all direct calls which target the callee. Figure 3.5 shows an example of optimized functions. function A and B are renamed to A.regx16 and B.regx16. The name of additional wrapper functions are same as their original function. A function calls from A to B, is replaced to B.regx16 to eliminate the redundant mode switch in wrapper function B.

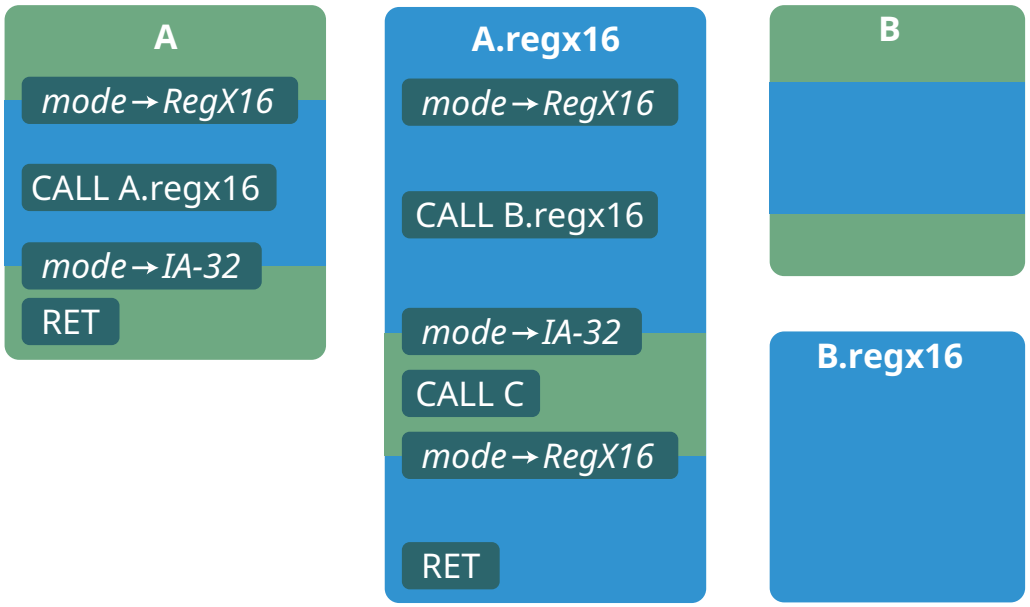



Figure 3.5: Optimized direct function calls without redundant mode switches

**Indirect Call Problems**

An indirect call in x86 uses a register to specify the address of the target function. We cannot eliminate mode switches of such function calls because modes of the targets are variable. However, in some cases the targeting functions are determined in compile time and stored as a constant list. If all the targets are RegX16 functions, it is safe to remove the switches.

Besides, because the mode is switched to IA-32 before the call instruction is executed,


doi:10.6342/NTU201603161

extended registers cannot be used here. The same problem occurs with the indirect call instructions of which the address is placed in memory. In these instructions, the memory location cannot be addressed by extended registers either. Therefore, we replace the register classes of such instructions by GR32_NOREX to avoid assigning extended registers to them.

**Optimizing Tail Calls**

The optimization for direct calls between RegX16 functions cannot be applied on tail calls since they do not return and we have no chance to switch the mode back. For example, assumed that the functions A, B are RegX16 and function C is IA-32. If A calls B normally and B calls C by tail jump, the mode is still IA-32 after function B returns, which is incorrect. Therefore, tail call optimization should be disabled when calling IA-32 function, but it is safe to use if the jump targeting RegX16.



doi:10.6342/NTU201603161

# 4 Experimental Results

In this chapter, we conduct the performance evaluation of our design and implementation for the mixed-mode execution of IA-32 and RegX16 code. Before the evaluation of mixed-mode execution, we also measure the performance improvement of pure RegX16 binaries, which are fully compiled to RegX16 and do not have mode switching.

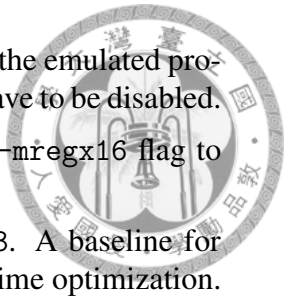
## 4.1 Environment

Our RegX16 processor is emulated by a FPGA board, which is about 250–300 times slower than modern PC-grade consumer processors. Therefore, we use the EEMBC benchmark set [1] for evaluations, since these benchmarks are lightweight enough to be executed within reasonable turnaround time using our FPGA board based system. The operating system used is Fedora, a Linux based open source OS. All benchmarks from EEMBC are compiled with our compiler, which is based on clang-3.8 and supports mixed-mode binaries of RegX16 and IA-32.

To evaluate the performance of RegX16, not only the benchmarks, but we also extract necessary functions from C libraries and compile them to RegX16. Therefore, mode switching between functions is not required, and we only have to enable RegX16 at the entry points of the benchmarks.

To evaluate the mixed-mode execution, all benchmarks are compiled with all the six


doi:10.6342/NTU201603161



| | |
|---|---|
| **Base** | Compiled with `-m32 -mno-sse -O3`. Since the emulated processor does not support SSE, these features have to be disabled. |
| **RegX16** | Same as the baseline but add an additional `-mregx16` flag to enable mixed-mode code generation. |
| **BaseLTO** | Compiled with `-flto -m32 -mno-sse -O3`. A baseline for other configurations which also enable link-time optimization. |
| **RegX16LTO** | Compiled to mixed-mode code with LTO support. |
| **SchedLTO** | Compiled with `-flto -m32 -mno-sse -O3 -march=rdc` to enable our customized scheduler and LTO support. |
| **SchedRegX16LTO** | Compiled to mixed-mode code with LTO and scheduler support. |

Table 4.1: Compilation configurations for evaluations

configurations listed in table 4.1. However, routelookup, bezier01fixed, and bezier01float benchmarks have been fully optimized away when LTO is enabled, and therefore, these benchmarks are removed from our testing set. Besides, we impose some additional restrictions on extended registers to some instructions for the ospf and cjpeg benchmarks to avoid triggering a hardware bug and crashing the benchmarks. The bug is that some extended registers cannot be used as the index of the effective address. Therefore, the real improvement of these two benchmarks is constrained by the lack of extended registers.

## 4.2 Performance Improvement of RegX16

We first evaluate the performance improvement of exploiting extended registers in RegX16. All the benchmarks are fully compiled to RegX16 and not linked with IA-32 libraries. The necessary library functions are extracted and compiled to RegX16. Therefore, the overhead of mode switching is excluded in this case. Figure 4.1 shows the improvement, which is based on the performance of original IA-32 binaries. Floating-point benchmarks, matrix01, basefp01, iirflt01, tblook01, and aifirf01, are removed in this experiment, since they certainly do not benefit from extended registers on integer operations. Some other


doi:10.6342/NTU201603161

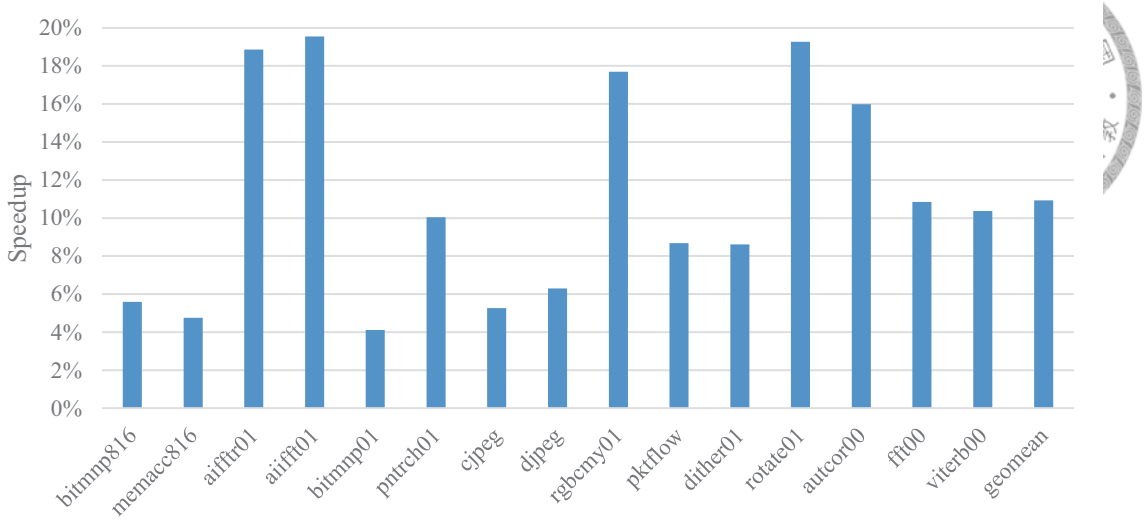



Figure 4.1: Performance improvement of RegX16

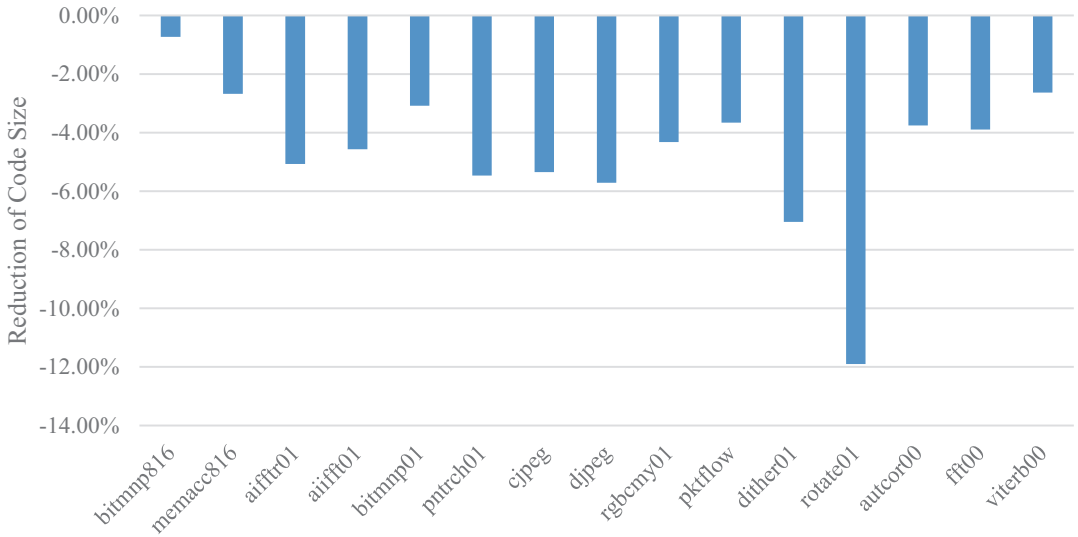



Figure 4.2: Reduction of code size when compiling to regX16

benchmarks are removed because their performance are not improved even if compiled to Intel64, which means that there is no performance opportunity by exploiting more registers. When the benchmarks are compiled to RegX16 and no mode-switching is required, the average performance improvement is 10.9%, and the performance of some benchmarks are improved more than 18%.

The comparison of code size of IA-32 and RegX16 is showed in figure 4.2. Though the length of some instructions may be increased by REX prefix, the total code sizes



doi:10.6342/NTU201603161

are reduced in all benchmarks since the number of instructions of register spilling and restoring is decreased.

## 4.3 Performance Improvement of Mixed-Mode

Figure 4.3 shows the performance improvement of each benchmark when compiled with each of the configurations. The performance of the Base configuration is used as the baseline performance of all other configurations. The result shows that the performance can be improved by extended registers in mixed-mode. When all optimizations enabled by compiling with SchedRegX16LTO, the average performance gain is 34.4%, which is better than SchedLTO without RegX16, 21.2%. The comparison of RegX16LTO and BaseLTO is similar, where the speedup of RegX16LTO, 22.5%, is larger than using LTO alone, 16.6%.

The pure RegX16 configuration is one exception, whose performance is worse than the baseline on average. In some benchmarks, the performance is even reduced more than 50%. The reason is that the frequencies of external function call in these benchmarks are much higher than others. We calculate the number of function calls in each benchmark by instrumentation and classify them into 3 kinds: external, indirect, and internal. The term “external” means to call a function that out of the module of the caller at compile-time; and “internal” means to call a function in the same module, respectively. Figure 4.4 shows the result. Since only the mode switching of internal and indirect calls can be eliminated without LTO support, those benchmarks with lots of external calls, canrdr816, memacc816, rspeed816, cacheb01, canrdr01, rspeed01, and pktflow, perform poorly when compiled with the pure RegX16 configuration.

On the other hand, if a benchmark does not frequently call external functions, it could



doi:10.6342/NTU201603161

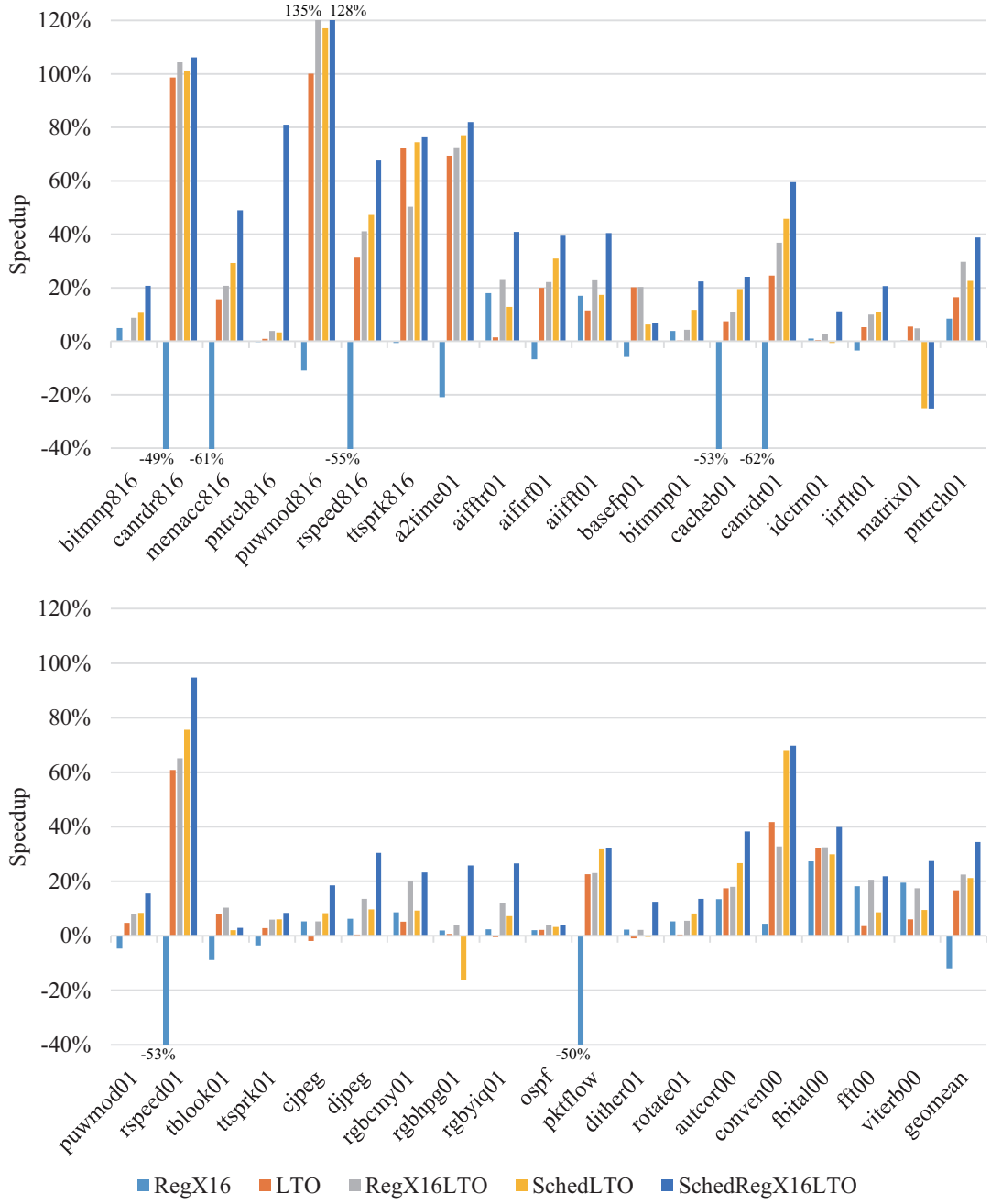



Figure 4.3: Performance improvement of all configurations



doi:10.6342/NTU201603161

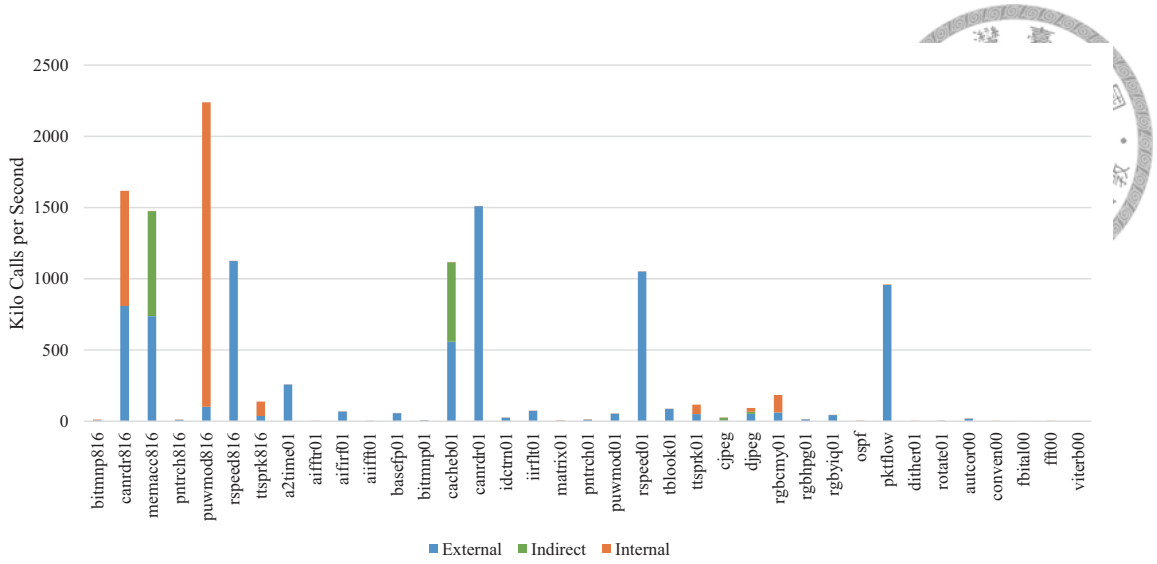



Figure 4.4: Frequencies of function calls

often take advantage of RegX16. For example, aifirf01, aiifft01, fft00, and viterb00 are improved by about 17%, and fbital00 has the maximum improvement, which is 27.3%. However, this is not always true since not all benchmarks could benefit from lots of general purpose registers. For example, basefp01 and maxtrix01 are floating-point applications, which requires more FP registers rather than the extended general purpose registers.

LTO accounts for a significant part of the performance gain from RegX16LTO and SchedRegX16LTO. Canrdr816 and puwmod816 gain about 100% speedup; ttsprk816, a2time01, and rspeed01 have over 60% speedup. The reason is that when LTO is enabled, most of the functions from separated modules are available to be inlined, and this inlining further increase the impact of other optimizations due to the enlarged optimization scope. Therefore, to measure the true performance gain with extended registers in mixed-mode, we have to use BaseLTO as the baseline of RegX16LTO and SchedLTO for SchedRegX16LTO. The result is showed in figure 4.5. The average speedup of RegX16LTO and SchedRegX16LTO are 5.1% and 10.9%. The best one in RegX16LTO is aifftr01, which has 21.2% speedup. Scheduled code delivers higher performance than non-scheduled because that to get full benefit of extended register requires the code scheduler to aware


doi:10.6342/NTU201603161

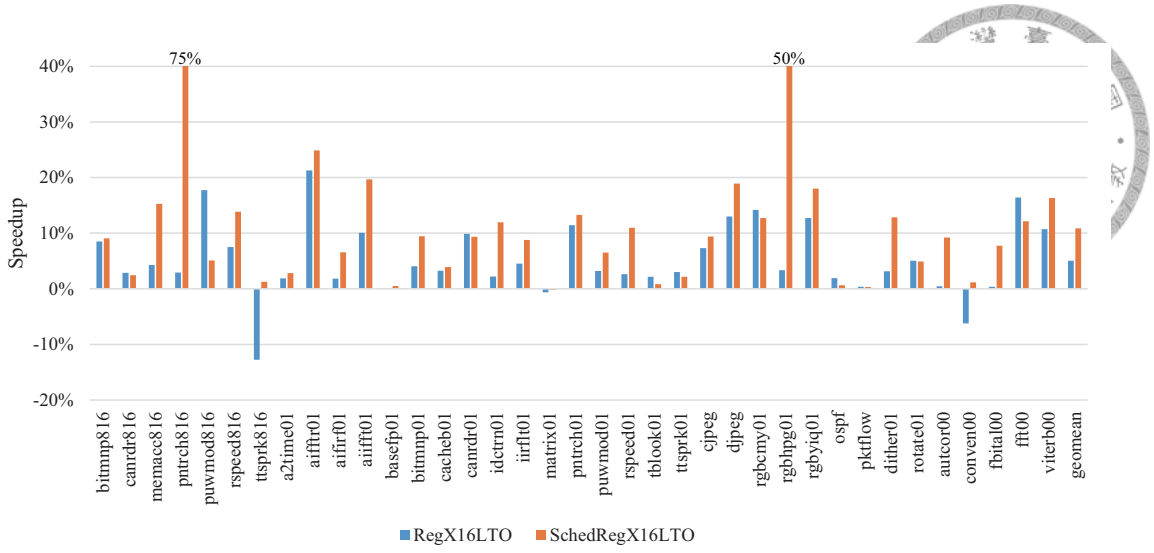



Figure 4.5: Improvement of exploiting extended registers

of the newly available registers. For example, RegX16 does not perform well for non-scheduled code in pntrch816 and rgbhpg01, but the performance is dramatically improved to 75.0% and 50.1% in the scheduled versions.

Instruction scheduling becomes more effective when RegX16 is enabled. The speedup of applying instruction scheduling on pure IA-32 code and mixed-mode RegX16 code is showed in figure 4.6, which use BaseLTO and RegX16LTO as baselines, respectively. On average, the improvement is 3.9% in IA-32 code and 9.7% in mixed-mode RegX16 code, which means that we have more effective instruction scheduling with extended registers.

Some benchmarks are slowed down when the instruction scheduling is applied. This is because that our scheduler aims to maximize ILP with some greedy heuristics, which does not always reach a perfect balance between ILP and limited registers. Over-scheduling could result in register spilling while under-scheduling may leave some bubbles in the pipelined execution. A more balanced scheduling algorithm could address this issue better. We leave this issue for future improvement.



doi:10.6342/NTU201603161

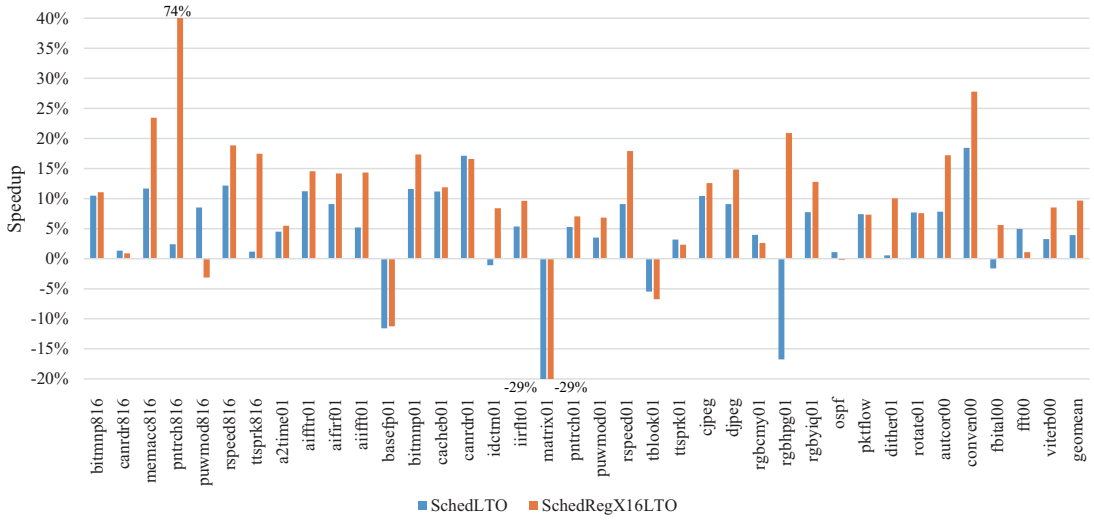



Figure 4.6: Improvement of instruction scheduling



doi:10.6342/NTU201603161

# 5 Conclusions and Future Work

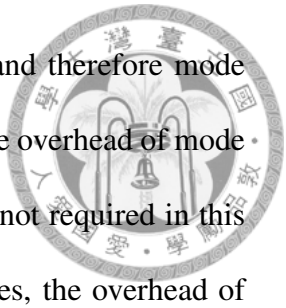
In this thesis, we propose a software-based mode switching mechanism to support mixed-mode execution. With our mechanism, instructions from two incompatible ISA can be mixed and executed in a binary without special hardware support. We can link our application binaries, which are compiled to RegX16, with legacy IA-32 libraries. Therefore, to make an existing application available to exploit the extended registers, we only need to recompile the applications themselves.

Our compiler implementation is based on LLVM and support code generation for mixed RegX16 and IA-32 binaries. Since RegX16 is directly extended from IA-32, we can leverage the existing IA-32 backend in LLVM to avoid tedious and error prone development work. Most of the source codes in IA-32 backend can be reused in the RegX16 backend, but we have to carefully remove unsupported IA-32 features from our backend to prevent generating illegal instructions in the RegX16 code. Our compiler can generate statically-linked, pure RegX16 binaries without mode switching or dynamically-linked, mixed-mode binaries of RegX16 and IA-32, and both executable types can be executed in our environment, which is an FPGA board with a Linux-based operating system.

The results of our experiments show that the performance of existing applications can be improved by recompiling to RegX16, either in pure or mixed mode. The performance of the benchmarks which may benefit from using more registers are improved by 10.9% in pure mode on average, and in mixed mode, the average speedup of all benchmarks



doi:10.6342/NTU201603161



is 5.1%. In pure mode, all source codes are compiled to RegX16 and therefore mode switching is not required. The performance is not encumbered with the overhead of mode switching. Optimizations for reducing redundant mode switches are not required in this case. However, if the application is compiled to mixed-mode binaries, the overhead of mode switching may affect the performance, especially when the source codes are full of external calls. Our optimizations can only eliminate redundant internal calls, but if link-time optimization is enabled, we can further eliminate external calls between source files. However, external calls which targeting library functions are required and cannot be eliminated.

Our experiments also show that the effectiveness of code scheduling can be improved with additional registers. When applying to RegX16 binaries, the performance improvement of code scheduling is larger than applying to IA-32 binaries. On the other hand, using extended registers with code scheduling is better than without it. Extended registers and code scheduling are benefit from each other.

Our software-based mode switching mechanism is fully independent to the mixed ISAs, which means that we can mix any two ISAs in a binary, not only the extended and the original one. One future work could be mixing ARM and x86 on Atom processors, which execute ARM binaries by binary translation. Our target is to reduce the overhead of binary translation by combining multiple binaries and executing them in mixed-mode if the future processors provide mode switching support.



doi:10.6342/NTU201603161

# References


[1] EEMBC - Embedded Microprocessor Benchmarks. `http://www.eembc.org/benchmark/products.php`. (Accessed on 08/15/2016).

[2] writing an llvm backend —llvm 3.8 documentation.

[3] Intel Corporation. *Intel® Itanium Architecture Software Developer's Manual*. October 2002.

[4] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253669-033US. December 2009.

[5] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Mar 2004.

[6] RDC Semiconductor Co., Ltd. Official website. `http://www.rdc.com.tw/`, 2016.

[7] David Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.




doi:10.6342/NTU201603161