國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

一個為 Thumb-2 可執行檔
以 LLVM 為基準的靜態二元轉譯系統

An LLVM-based Static Binary Translation System
for the Thumb-2 Executable

研 究 生：劉冠宏

指導教授：徐慰中 教授

中 華 民 國 **102** 年 **7** 月

一個為 Thumb-2 可執行檔以 LLVM 為基準的靜態二元轉譯系統

An LLVM-based Static Binary Translation System for the Thumb-2 Executable

研 究 生：劉冠宏　　Student：Kuan-Hung Liu

指導教授：徐慰中 博士　　Advisor：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 Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

In partial Fulfillment of the Requirements

for the Degree of

Master

In

Computer Science

July 2013

Hsinchu, Taiwan, Republic of China

中華民國 102 年 7 月

一個為 Thumb-2 可執行檔
以 LLVM 為基準的靜態二元轉譯系統

研究生：劉冠宏　　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

Thumb-2 是一個 16 位元和 32 位元共存的指令長度可變指令集架構，跟 ARM 架構相比，他有更高的指令密度，但是效能又很接近 ARM。對靜態二元轉譯系統來說，如何區分指令和資料，以及找到轉譯前後程式計數器的對應是非常困難的，因此設計一個靜態二元轉譯系統不是一件簡單的事情。在這篇論文中，我們介紹一個對於 Thumb-2 可執行檔的靜態二元轉譯系統，它利用了 LLVM 的各項功能去轉譯輸入的檔案、對他做最佳化、編譯，並且產生輸出的二進位檔。我們的系統利用一些方式找到那些被 GCC 所產生出來的二進位檔中，被安插在指令間的資料，而且建立了一個轉譯前後程式計數器的對應表並利用一些方法減少此表的空間。我們亦提供了一些方法改善我們轉譯後的檔案，使得 LLVM 優化器和編譯器可以更快的完成他們的工作。我們的系統最終產生 x86 架構的可執行檔以便於比較效能，並使用 SPEC2006 CINT 配合具參考價值的輸入資料來做為比較的依據，就平均的結果來看，我們轉譯後的可執行檔比使用 QEMU 的結果快了大約 5.6 倍；而跟 x86 原生的可執行檔比較起來，速度大約慢了 2.1 倍，且檔案大了 2.5 倍。而最後我們提出的一個減少工作時間的方式雖然執行時間多花了三成，可是轉譯的時間卻快了 13 倍。

An LLVM-based Static Binary Translation System for the Thumb-2 Executable

Student: Kuan-Hung Liu Advisor: Dr. Wei-Chung Hsu

Degree Program of Computer Science
National Chiao Tung University

ABSTRACT

Thumb-2 is a 16-bit and 32-bit mixed instruction set architecture (ISA), with higher code density compared with ARM, and the performance is close to ARM. The code discovery problem and the code location problem caused by indirect branches make static binary translation (SBT) system hard to develop. In this thesis, we present a SBT system for Thumb-2, which leverage the LLVM infrastructure to translate the source binary into LLVM IR, optimize and compile the LLVM bitcode file, and then generate the target binary. Our system solves the code discovery problem for the binaries, which are generated by GCC, by finding all kinds of data that are interspersed in the code. The code location problem is also solved by creating an address mapping table with relatively smaller size. We also introduce an approach to reduce the optimization and compilation time of translated LLVM bitcode files. Our system finally generates x86 executable for performance comparison. In our experiments which use SPEC2006 CINT with reference data to be the benchmark, the execution time is about 5.6 times faster than QEMU, while about 2.1 times slower with 2.5 times code expansion when compared with the x86 native binaries. Furthermore, with our saving-time approach, the execution time will be increased by 30% while the translation time could be 13X better.

誌　　謝

本論文能夠順利完成，首先必須感謝我的指導教授徐慰中老師，老師帶給我們的不只是專業領域的知識，還包括許多寶貴的人生經驗和處世態度，都值得讓未來的我們作為參考，甚至訂為目標；更感謝三位口試委員：吳真貞教授、單智君教授、楊武教授，不但撥空前來參與我的口試，他們的意見和指教也讓本論文更加的充實和完整，由衷的感謝他們。也謝謝我的前指導教授莊榮宏老師，願意在我對計算機圖學失去興趣的時候讓我轉換領域，讓我得以在新的領域貢獻一點微薄的心力。

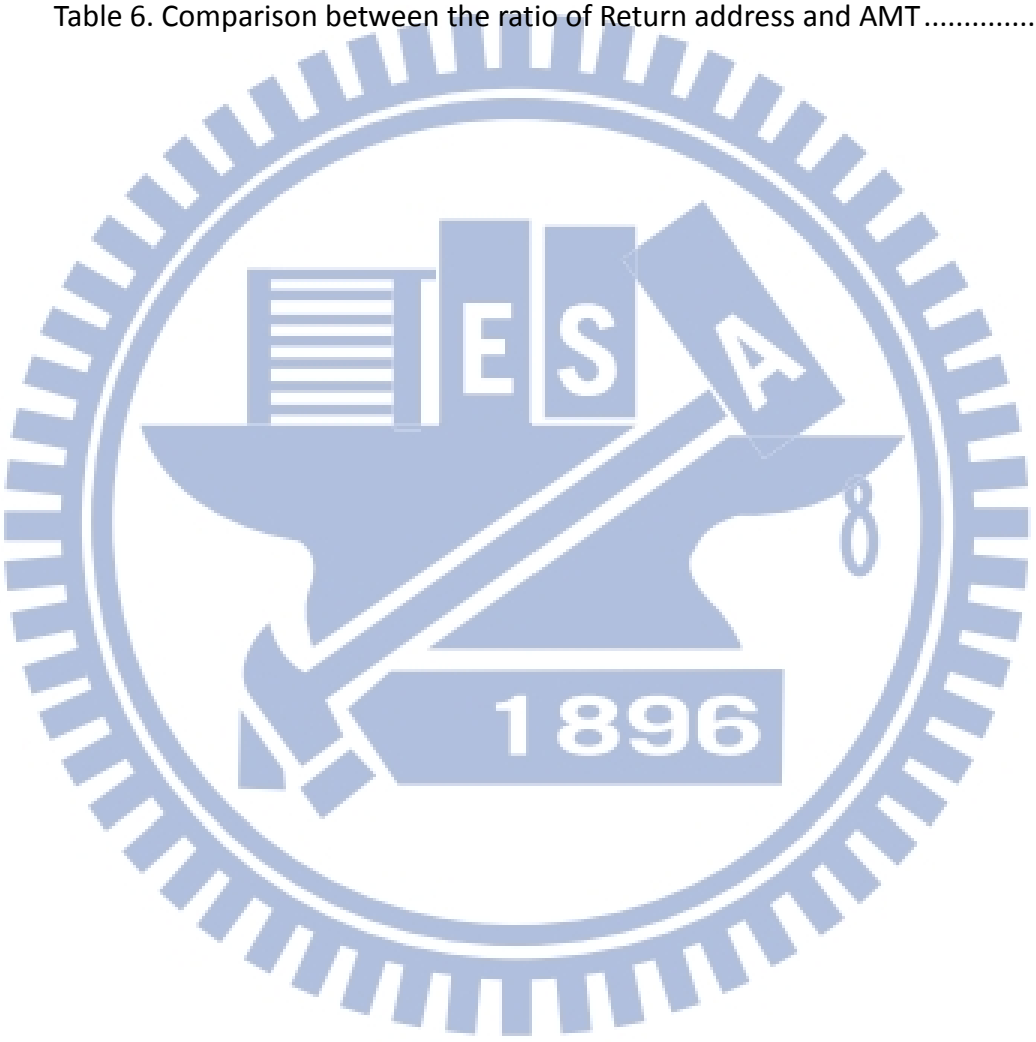
特別感謝陳俊宇學長和沈柏曄學長在研究上的指導，幫助我突破許多論文上的瓶頸，讓本論文能夠順利完成；感謝所有 446A 實驗室和計算機圖學與幾何模擬實驗室的同學、學長姊和學弟妹及助理們，碩士的兩年有你們陪伴，讓我過得很充實也很愉快，那段一起研究如何泡咖啡的日子，我永遠也不會忘記，真的非常謝謝你們。還要感謝中華扶輪教育基金會，我得到的不只是金錢上的援助，讓我得以安心念完碩士，更重要的是，讓我在獎學生聯誼會中，認識一群各領域的菁英，同時也是一群值得信賴的夥伴，能順利完成論文，他們也功不可沒!此外，每個月的例會更讓我在忙碌的研究生活中，找到一個喘息的機會，有好多平常沒有機會到訪的地方，都藉由參加例會一一的實現，由衷的感謝那群支持基金會運作的扶輪社友們以及聯誼會的幹部們。

最後，感謝一直以來支持我、為我操心的家人，讓我一路順利且無後顧之憂的念完碩士，你們的支持是我最大的力量，也讓我更有勇氣去面對未來的困難。

謝謝所有曾經幫助過我、關心過我的人，在此表達我最誠摯的謝意，有了你們的支持，未來的我會更加努力。

Table of Contents

List of Tables



List of Figures

I. Introduction

Binary translation [1] techniques have been used in various areas, such as application migration from one ISA (Instruction Set Architecture) to another [2] [3] [4], fast simulations [5], dynamic optimizations [6] [7], and virtual machine implementations [8]. These techniques have been actively studied and developed in the past decade.

Software-based binary translation can be roughly classified into two categories: SBT (Static Binary Translation) and DBT (Dynamic Binary Translation) [9]. Both of them have pros and cons, but most of production systems and researches are based on DBT. This is because two major problems of SBT, code discovery problem and code discovering problem, can be more effectively and efficiently handled by DBT. DBT has some shortcomings, such as longer start-up time due to initial runtime translation, less aggressive code optimizations for translated binaries compared with SBT, and larger memory footprint due to the use of code cache, the emulation engine and the presence of the dynamic translator. In some environments, like the embedded systems, that start-up time, power consumption, and memory usage are the primary concerns, using SBT might be more desirable than DBT. Some researcher works offer HBT (hybrid binary translation) [10] that combines the advantages of both SBT and DBT. It uses SBT to translate the guest code as much code as possible statically, and switches to DBT when run-time exceptions occur due to incorrect translation by SBT. Ideally, HBT has the advantage of high performance due to SBT, and the robustness of DBT. However, if the code discovery and code location problems are not effectively solved for SBT, then HBT could fall back to DBT and loses the advantages of SBT. In this thesis, we focus on solving the two problems in SBT for the ARM/Thumb-2 architecture.

ARM [11] [12] architecture is widely used in the mobile computing market and embedded systems. More and more companies adopt ARM processors in their systems, including most smart phones, tablets (including MS Windows RT), and Google Chrome-Book laptops. Recent ARM cortex A15 may start to show up in micro-servers. The original ARM is a RISC (Reduced Instruction Set Computer) ISA, all instructions are fixed-length. However, due to the memory footprint requirements for embedded systems, the ARM architecture has been enhanced with variable length instructions in ARM/Thumb-1 and ARM/Thumb-2.

All instructions in Thumb-1 are 16 bits, and there is a mode bit that indicating the current execution mode is ARM or Thumb-1 for ARM/Thumb-1 mixed mode binaries. To further reduce the code size and increase the performance, Thumb-2 ISA was introduced after ARMv6T2. This thesis concentrates on the translation of Thumb-2 instructions since it achieves performance similar to ARM code, with code density close to Thumb-1. The difficulty for translation is that Thumb-2 is considered a mixed length (or variable length) ISA which contains both16 bits and 32 bits instructions. With a variable length instruction ISA, code discovery problem becomes an issue. To solve the code discovery problem in the Thumb-2 binary translation, we have to identify all kinds of data embedded in the binary. Data embedded in binary can be PC-relative data, instruction padding and jump/search tables generated from the switch statements. Identify the jump and search tables is challenging, since the code patterns generated from different compilers are not the same. This issue is investigated in this work. In the past, our team developed LLBT [4], and has dealt this problem for ARM and Thumb-1 binaries generated by GCC [13], which is one of the most popular compiler used in the Linux world.

Once the binary is translated to the target-binary, it should be optimized; otherwise, the performance is too poor to be accepted as a desired alternative to DBT. Developing a binary translator that includes decoder, translator, optimizer, assembler, and linker takes too much efforts and these work usually cannot be leveraged when the target is changed, due to the nature of high target-machine-dependence. Therefore, a retargetable binary translator is highly desirable. To implement a retargetable binary translator, the translation pass must be split into target-independent part and target-dependent part, and the interface between the two parts are immediate representation (IR) forms. The front-end of our SBT system translates the input binary into IR forms and then translates into the binary of target machine with different ISA by the back-end. Instead of creating a new IR, we lean to leverage an existing compiler infrastructure that can satisfy our requirements to build a high-performance and retargetable binary translator.

In this thesis, we present a SBT system that leverages the LLVM [14] infrastructure and can translate Thumb-2 ISA to many different target ISAs supported by LLVM. Our work is based on mc2llvm [10], a retargetable hybrid binary translator.

The remainder of the thesis is organized as follows: Chapter II introduces some related work and the techniques used in our work. Chapter III gives the design of our system and describes the implementation details. Chapter IV discusses the experimental results and Chapter V concludes our work and discusses future works.

II. Background and Related Work

In this chapter, we introduce binary translator first, and then describe the main problems we met when constructing a SBT system. Thumb-2 instruction set is also introduced in this chapter. Previous works on solving the code discovery problem are also included. An overview of the LLVM compiler infrastructure is then described. At the remainder of this chapter, we briefly show how mc2llvm, which is the base of our system, works, and introduce some techniques that are also important for our system.

2.1. Binary translator

Binary translator translates the source binary program into the target binary program whose ISA may be different from the source one. It can be classified into two categories: SBT and DBT, and they translate the source binary off-line and on-line respectively.

2.1.1. Static Binary Translator

Static binary translator translates the source binary into the target binary in static time, so there is no translation overhead when executing translated binary. Besides, more optimizations can be applied to SBT since the time that costs in compiling time is not that important. Therefore, the performance of the executable generated by SBT is always better than using DBT. However, the programmer must solve the code discovery problem and the code location problem, which are known as the most critical problem when constructing a SBT.

Chen et al. [2] built an SBT that translates the ARM binaries into a MIPS-like platform without using target independent IR. Since all of the ARM instructions are 32-bit instruction, the translator can still work correctly without solving code discovery problem. Their translator uses an address mapping table to solve code location problem. Furthermore, their translator has a restriction that only the binary generated by GCC can be translated correctly since their translator cannot ensure that it can find all of the indirect branch targets in the binary generated by the compiler other than GCC. This restriction is the same as our system, although our purpose is different.

2.1.2. Dynamic Binary Translator

Dynamic binary translator translates the source binary at run-time, and puts the translated code in the code cache. An instruction is translated only if the control flow reaches it, so DBT won't waste any time to translate unused instructions. Therefore, the programmer don't have to solve the code discovery problem since the control flow never reaches data sections. However, the optimizations that can be applied in DBT is much less than is SBT, because the performance will also be influenced when optimizing.

QEMU [15] is an efficient and retargetable DBT which supports both user-mode and full-system emulations. It is widely used since many popular ISAs are supported and it is an open-source software. QEMU translates the source binary instructions into a sequence of micro operations which are implemented by a small pieces of C code. Then these C code will be compiled into the target binary by GCC. Newer version QEMU uses tiny code generator (TCG), which provides a small set of operations, to parse the micro operations and generates the target binaries.

2.2. Code Discovery Problem

Code discovery problem [16] is really a trouble when constructing a SBT system because it may lead wrong results. The reason why this problem occurs includes variable-length instructions, register indirect jumps, data interspersed with the instructions, and padding bytes to align instructions, as shown in Figure 1.

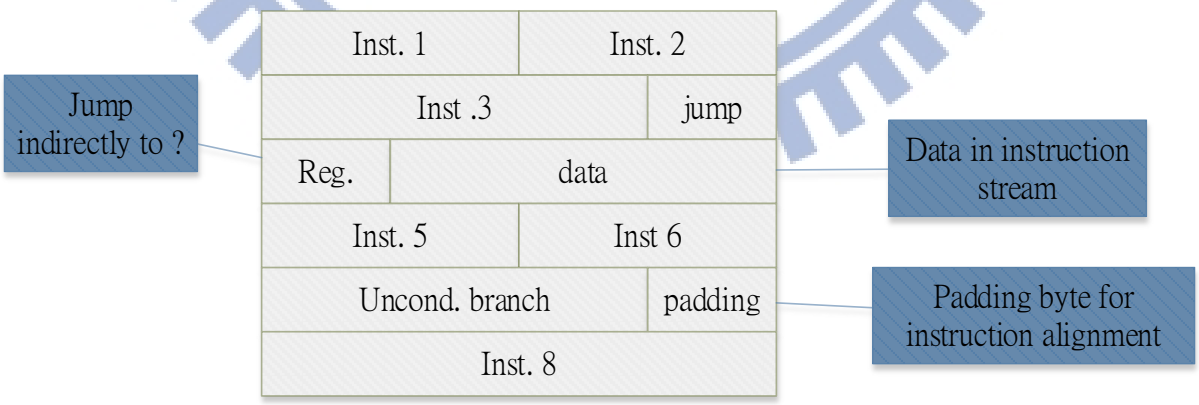


Figure 1. Causes of the Code Discovery Problem

2.2.1. Variable-length Instructions

If the instructions are fix-length, then all of the instructions can be translated correctly.

Even if data or padding bytes are regarded as an instruction, there is no control flow reaches them. Therefore, misclassification of code and data in the fix-length ISA performs no influence to the translator, except several dummy instruction blocks may exist in the target binary generated by the translator.

For variable-length instructions, the instruction boundaries may be difficult to find when data interspersed with the instructions. For example, as shown in Figure 2, the instruction that is disassembled from different start address is different. 0x2000 is a MOV instruction but 0x6220 is STR instruction, so the programmer should handle it carefully when designing a SBT.

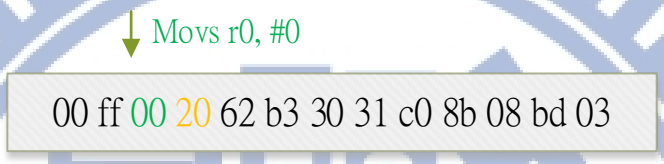


Figure 2. An example of finding Thumb-2 instruction boundaries

2.2.2. Register Indirect Jump

When an indirect jump instruction is encountered, the target address of this jump instruction is held in the register. Determining the content of the register in static time is very difficult, because some of the values are not decidable until run time. Moreover, deciding whether the instruction immediately following the jump instruction is valid is also difficult, since it can be a part of data or padding bytes.

2.2.3. Data Interspersed with the Instructions

Some ISAs allow data interspersed with the instructions, and it makes the SBT more difficult to construct because the data may be regarded as instructions. The kinds of data that may be interspersed with instructions are PC-relative data, switch tables, searching tables … and so on. The manual of ISAs may provide some information about how these kinds of data are usually dealt with. Usually the most difficult reason that the code discovery can't be solved is because these data can't be found exactly.

2.2.4. Padding Bytes to align Instructions

Different ISAs have their own alignment restriction. For example, ARM instruction set

must be word-alignment while Thumb-2 instruction set must be halfword-alignment. This kind of bytes may be more difficult to find when using CISC ISAs, like x86. Although some compilers use an NOP (no operation) instruction to implement padding behavior and the programmer can regard them as an instruction, padding bytes with value being certain value is still more common.

2.3. Code Location Problem

Since the source binary is accessed by the source program counter (SPC), while the translated binary is access by the target program counter (TPC), the problem occurs when there is an indirect branch or jump in the source binary. The target address of indirect control transfer is stored in the register and this address is belong to the source binary. The address of the translated binary may be different from the source. Therefore, a SPC address to TPC address mapping is needed; otherwise, the target address for translated binary is still unknown and the result is unpredictable.

2.4. ARM/Thumb-1 mixed ISA

ARM is a 32-bit ISA while Thumb-1 is a 16-bit ISA. Thumb-1 is introduced because many ARM instructions need only 16 bits to encode and it wastes many code spaces. However, the performance of Thumb-1 executable is worse than ARM executable, since more Thumb-1 instructions are needed for performing the same efforts as one ARM instructions. Before Thumb-2 instruction set was introduced, the only way to leverage the pros of these two ISAs is using ARM/Thumb-1 mixed ISA.

ARM/Thumb-1 mixed ISA uses a mode bit to indicate what the current ISA mode is, and the bit 0 of PC to indicate the ISA of branch target region. However, the content of this mode bit is known at run time, and the translator can't decide which kind of ISA is used for encoding in the current region, which starts at certain address and ends with a branch instruction, in static time. Fortunately, both of ARM and Thumb-1 are fix-length, so, as we mentioned in 2.2.1, even though the translator regard some data as instructions, it won't influence the correctness of the translated result. As the result, the code discovery problem in ARM/Thumb-1 ISA is reduced to find which regions should be disassembled as ARM instructions and which should be Thumb-1.

Chen et al [17] introduce a method to distinguish what kind of ISA the current region of instructions were encoded. Their system translates the input binary using ARM and Thumb-1 ISA respectively and gets two version of translated code. Instead of exactly choosing the correct one, their purpose is to discard the regions that must be error-translated. They define “safe region” as the region that is decidable, that is, it must be certain ISA. Safe regions can be found by the entry point of the program from ELF file, head address of each sections, and sections that contain function constructors and destructors. A region of branch targets of safe regions is also a safe region, so many safe regions can be found. Nevertheless, two regions may be linked together by an indirect branch, so there are still a lot of regions that are unknown. Unknown regions can be analyzed and be discarded if some illegal situations occur. They definitely translated an ARM/Thumb-1 executable effectively and correctly, since the code size of translated executable is only about 25% more than the best case and the performance is about 15% slower. This approach may useful when constructing an SBT system for ARM/Thumb-2 mixed ISA.

2.5. Thumb-2 Instruction Set

Thumb-2 is a 16-bit and 32-bit mix instruction set with higher code density and almost the same performance compared with ARM instruction set. All of the instructions in Thumb-2 binary are halfword-aligned, so the instruction can start with the address of multiple of two. Furthermore, Thumb-2 instructions are disassembled halfword by halfword, that is, the disassembler reads a halfword at a time, and decide whether it has to read next halfword by the first five bits of this halfword. Only the instructions start with 0b11101, 0b11110 or 0b11111 are regard as 32-bit Thumb-2 instructions. As a result, a continuous four bytes read from the binary are handled in different ways in Thumb-2 instruction set and ARM instruction set, although they are both 32-bit instructions, as shown in Figure 3. Besides, the instructions that take the same effect are decoded in different ways using these two instruction sets. Taking Figure 4 as an example, the left column is an ARM binary, and the right one is a Thumb-2 binary. The number of instructions and the efforts of this part of code of two binaries are equal, but the Thumb-2 one uses less space due to 16-bit instructions being used. Obviously, the encoding method of them are also different.

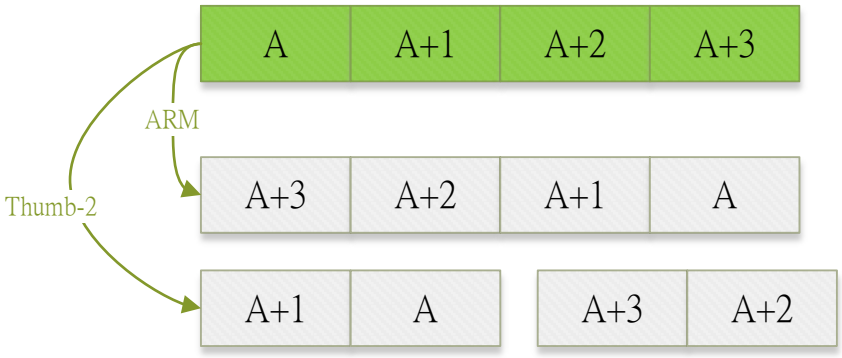


Figure 3. Comparison of 32-bit instruction between ARM and Thumb-2

```
000080d0 <_start>:
   80d0:   e3a0b000    mov fp, #0
   80d4:   e3a0e000    mov lr, #0
   80d8:   e49d1004    ldr r1, [sp], #4
   80dc:   e1a0200d    mov r2, sp
   80e0:   e52d2004    str r2, [sp, #-4]!
   80e4:   e52d0004    str r0, [sp, #-4]!
   80e8:   e59fc010    ldr   ip, [pc, #16]
   80ec:   e52dc004    str ip, [sp, #-4]!
   80f0:   e59f000c    ldr   r0, [pc, #12]
   80f4:   e59f300c    ldr   r3, [pc, #12]
   80f8:   ea000bd1    b <__uClibc_main>
   80fc:   eb000985    bl <__GI_abort>
```

```
000080c0 <_start>:
   80c0:   f04f 0b00    mov.w     fp, #0
   80c4:   f04f 0e00    mov.w     lr, #0
   80c8:   f85d 1b04    ldr.w r1, [sp], #4
   80cc:   466a         mov r2, sp
   80ce:   f84d 2d04    str.w r2, [sp, #-4]!
   80d2:   f84d 0d04    str.w r0, [sp, #-4]!
   80d6:   f8df c014    ldr.w ip, [pc, #20]
   80da:   f84d cd04    str.w ip, [sp, #-4]!
   80de:   4804         ldr   r0, [pc, #16]
   80e0:   4b04         ldr   r3, [pc, #16]
   80e2:   f002 b8c3    b.w <__uClibc_main>
   80e6:   f001 fdd9    bl <__GI_abort>
```

Figure 4. An example of ARM (left) and Thumb-2 (right) binaries

PC value in Thumb-2 instruction set is also different from what in ARM. The value is the address of current instruction plus four, while plus eight in ARM instruction set. This is because there are three pipeline stages: fetch, decode, and execute, in ARM7 CPU, which implements ARM architecture v4T and earlier; therefore the PC values in three stages are what are shown in Figure 5. For compatibility, PC value of ARM and Thumb ISA are defined as described above. Furthermore, since 16-bit Thumb-2 instructions use less bits, they may not use some register as an operand. For example, SP and PC are not permitted to use in many 16-bit instructions. Besides, Thumb-2 instruction set can use IT block, which describe the condition of at most four instructions following the IT instruction, to perform conditional execution if the instructions have no condition code bits. Some comparisons between ARM Thumb-1 and Thumb-2 instruction set are listed in Table 1.

ARM	PC	PC-4	PC-8
Thumb	PC	PC-2	PC-4

Figure 5. PC values in three pipeline stages ARM CPU

Table 1. Comparison between ARM, Thumb-2 and Thumb

	ARM	Thumb-1	Thumb-2
performance	high	low	close to ARM
Code density	low	high	
Instruction length	32 bit	16 bit	16 bit and 32 bit
Alignment	Word-aligned	Halfword-aligned	
PC	Current instruction address plus 8	Current instruction address plus 4	
Conditional execution	Condition code	IT block and condition code	
Get 32 bit instruction	A+3, A+2, A+1, A		A+1, A+0, A+3, A+2

Code discovery problem in Thumb-2 ISA is not the same as what in ARM/Thumb-1 mixed ISA, although they are both ISAs that contain 16-bit and 32-bit instructions. In ARM/Thumb-1 mixed ISA, the question is how to decide the ISA used for encoding current region, while how to find all kinds of data in Thumb-2 ISA. Moreover, some CPU has ability to run both ARM and Thumb-2 instructions, according to the value of the mode bit, which is the same as what is ARM/Thumb-1 mixed ISA; therefore, ARM/Thumb-2 mixed ISA also exists. To solve the code discovery problem in ARM/Thumb-2 mixed ISA, both deciding ISA of current region and finding data have to be solved, so both the approach in [17] and this thesis may be applied for this purpose.

2.6. Low Level Virtual Machine (LLVM)

LLVM [14] is an open source compiler framework that is developed by University of Illinois. It has ability to convert machine-independent instructions to machine-dependent assembly code. In addition to a static compiler, LLVM also includes Machine Code toolkit [18] which provides several tools for the relating works of the instruction set, such as assembler, disassembler, and object file handler … and so on. Several analysis phases and optimizations have been added to the LLVM infrastructure due to the rapid development in recent years.

LLVM IR is target-independent and must be SSA-form, that is, it can own unlimited virtual registers but each of them can just be defined once. Therefore, optimizations for these IRs can be performed no matter what the target ISA is and the programmer don't have to argue about the use of the registers.

2.7. MC2LLVM

Our SBT system is based on mc2llvm project, so we will introduce some design details of mc2llvm, especially the parts that also be used in our system.

Mc2llvm is a hybrid binary translator (HBT), which performs the same behavior as SBT in static time, except some routines for calling DBT when exception occurs. An overview of SBT part of mc2llvm is shown in

Figure 6. Since mc2llvm translates ARM binaries to LLVM IR, it don't need to find the data interspersed in the code. After translating the binary to LLVM IR, the LLVM optimizer and the LLVM static compiler are used for generating the target assembly code. Finally, the necessary files are linked together and then the target binary is generated.

Once the indirect branch target address cannot be found in the address mapping table, the program will switch to the DBT, which translates instructions from the address one by one until branch instruction occurs and the results are stored in the code cache. Then it switches back to original part of the program that is translated by SBT.

Mc2llvm uses an address mapping table to get the indirect branch target of the binary translated by mc2llvm. The entries in the table are function entries and return addresses, which are more likely to be the branch targets. A hashing function and LLVM switch instructions are used to construct this table, and the detail will be introduce later since our

system uses similar approach.

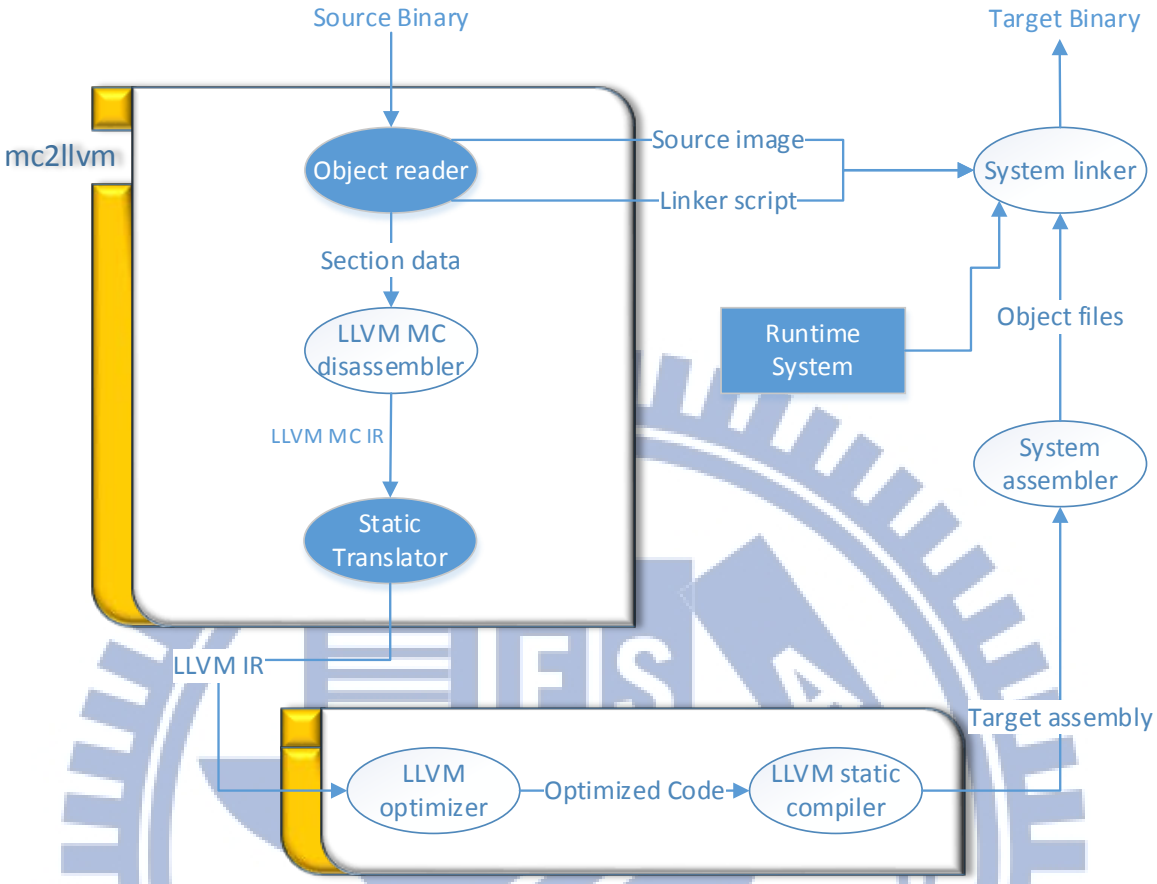


Figure 6. An Overview of SBT of mc2llvm

The function initialization routines, including allocating the stack, parsing the command line arguments, setting up the environmental variables, are written in LLVM IR, and this work is the same no matter what source ISA is, so our system uses this module without unnecessary modification. The system call emulator is implemented by several helper function written in C++ programming language. Since the Linux kernel system calls won't change if different ISA is used, we apply this emulator in our system, too.

For convenient, mc2llvm maps the memory address of the source binary to the target machine directly; therefore, the output binary uses the memory space beginning with 0x8000 to generate its own memory layout except the stack. As a result, the maximum address of the heap of output binary must be smaller than 0x8048000, which is address of read-only part when loading an binary for Linux system; otherwise, it cause segmentation fault. The memory layout of the target binary is shown in Figure 7. This problem does not exist when the environment is a 64-bit operation system with Linux kernel, since the memory space becomes much larger. This layout is also used in our system, except the source binary

is compiled using Thumb-2 ISA.

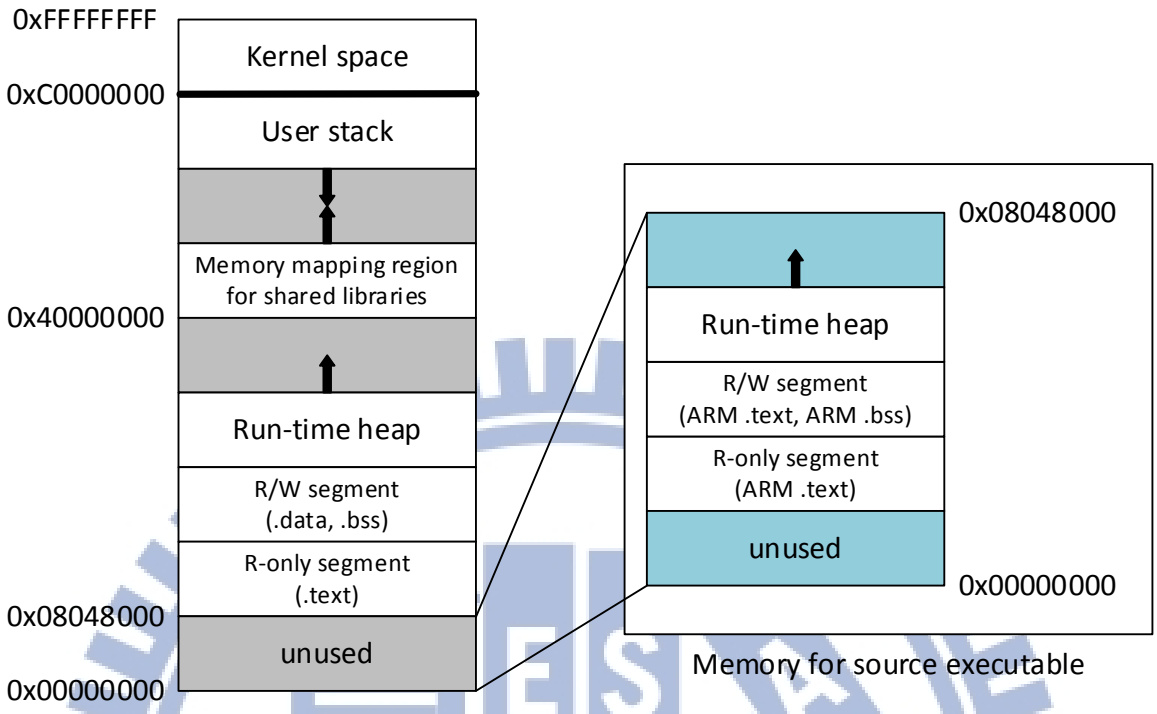


Figure 7. Memory layout of the target binary

III. Design and Implementation

In this chapter, we describe the framework of our static binary translator system first, and then introduce problems, including but not limited to code discovery problem and code location problem, we have to solve and have solved. Moreover, the data structure and methods that are used in our system will also be illustrated. Finally, we introduce some modifications we implemented for certain enhancement. At the remainder of this chapter, we give some discussion about how to relax the restrictions of our program to get a more general SBT.

For convenience, we define “B-function” and “L-function”, which indicate the function from the input executable and the LLVM function generated by our translator, respectively.

3.1. Overview

Our work is based on mc2llvm [10], which is a HBT, so the framework of our SBT system looks like the one in original mc2llvm.

Figure 8 shows the flow of our system. Our system uses LLVM API to read a source binary file, disassemble instructions, and translate them into LLVM IR. Summary of our system is shown as below:

1) Instructions are read by the object reader and decoded by the LLVM MC Disassembler, which is a part of LLVM MC Toolkit. The resource image is also generated in this step.

2) The analyzer analyze the binary and store the information about where is code, where is data, and some statistical data that are useful in the translator.

3) The static binary translator translates the MC IR generated from LLVM MC Disassembler into LLVM IR, and store the results to an LLVM bitcode file.

4) The LLVM optimizer (opt) is used to perform target-independent optimizations on bitcode file generated in 3).

5) The LLVM static compiler (llc) compiles the optimized bitcode file to target assembly, and performs some target-specific optimizations.

6) The target assembler assembles the target assembly and generated target object code.

7) The target linker reads the linker script generated by the object reader and links the resource image together with the target objects. A run-time system that includes a system call emulator and helper functions are also dynamically linked with the target binary.

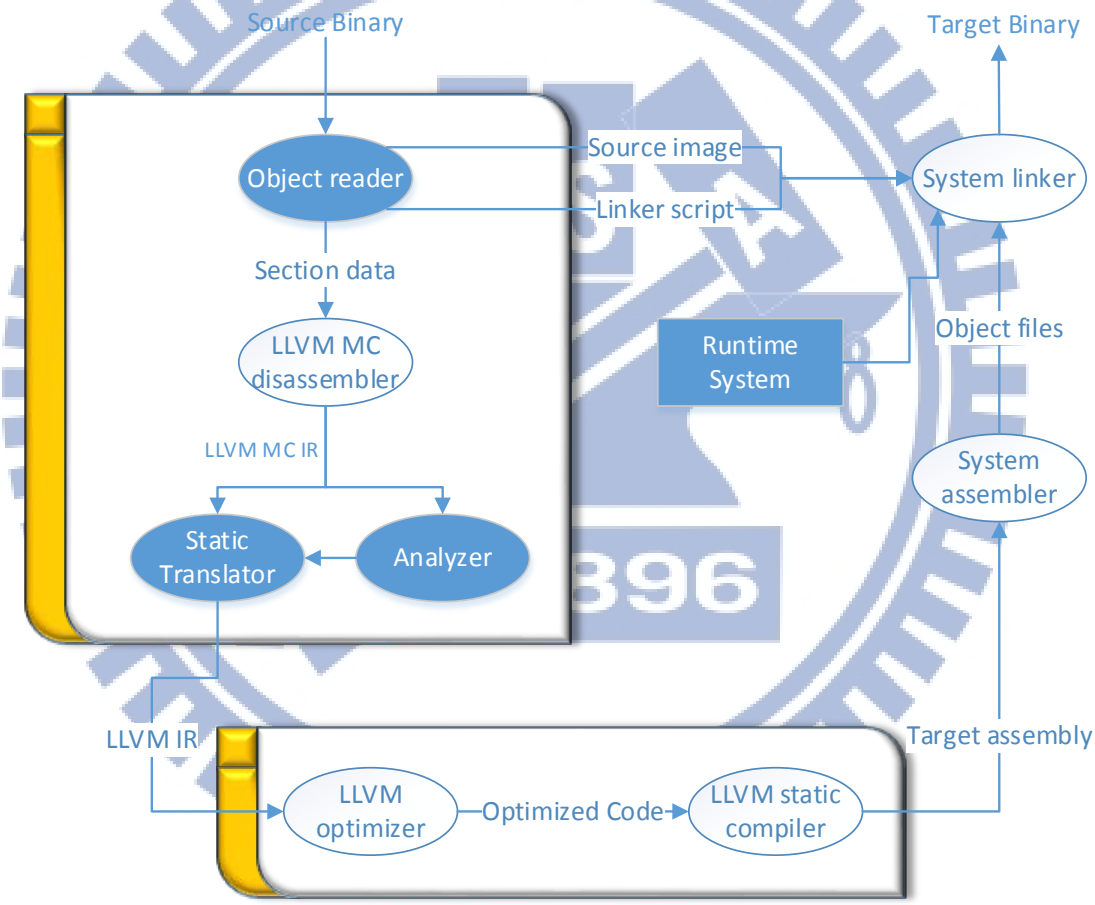


Figure 8. An overview of our SBT System

Figure 9 shows how the translated program, which is described by LLVM IR, works. Our translator adds some initialization routine at the beginning of the main LLVM function, including reading command line arguments, initialing the stack …etc. Instructions in the main L-function may call LLVM intrinsic functions or some user-defined helper functions. Every time the indirect branch occurs in the main L-function, the control flow switch to address

mapping stub, which load PC and jump to certain LLVM basic block with corresponding address. The translated program is terminated by some system calls.

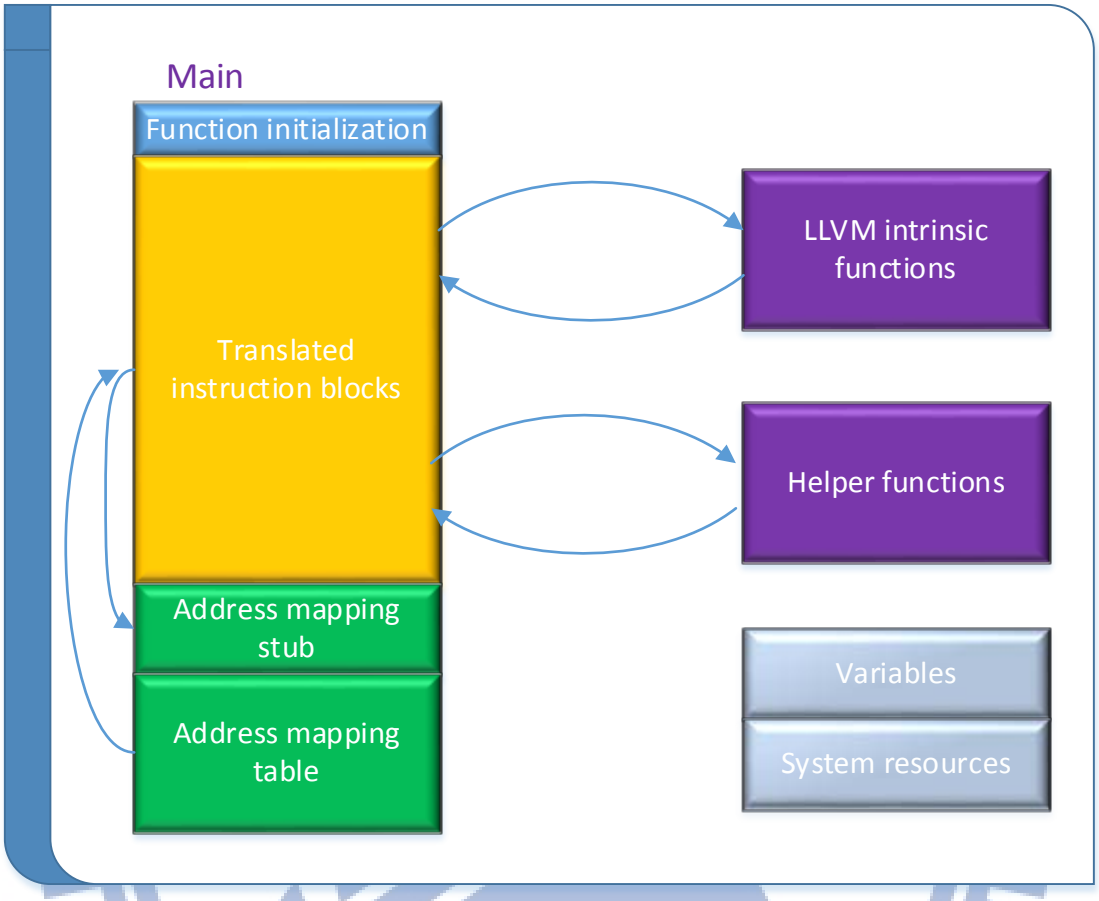


Figure 9. The framework of the translated program.

Our translator claims that a binaries that fit the constraints described below can be statically translated to LLVM IR by our translator.

1) Generated by GCC: we ensure that our translator find switch tables in the binary when certain patterns occur, and we just find the patterns that GCC may generate in our translator.

2) Use static linking: our system is an SBT, so it must be compiled using statically link; otherwise, exceptions occur when calling system libraries or all system libraries should be translated statically.

3) Single thread program: our translator doesn't support multi-thread and associated system call, so only single thread program can be translated correctly.

4) Use Linux kernel: the system call emulator in our system handles Linux system call

only.

3.2. Design Issues

Code discovery problem and code location problem are the main problem we have to solve. The former is due to we don't know where is the code and where is the data in the source executable, and the latter is due to the source indirect branch target address is not the same as the target indirect branch target address. For more detail, please see 2.2 and 2.3. We show how we solve these problems and describe other problems we encountered in this chapter.

3.2.1. Code Discovery Problem

As described in 2.5, distinguishing data and code in Thumb-2 binary is the major challenge we have to defeat when solving code discovery problem. Data interspersed with code in the binary make this problem hard to solve, so we classify all of data into several sets first and then conquer each set separately. These data can be classified into four kinds:

1) PC-relative data
2) Switch table
3) Padding
4) Unidentified cases

Once our translator finds all of them, it knows where the data are, and this problem is solved. We will describe how these data generated in Thumb-2 code and how our translator finds them in next section.

3.2.1.1. PC-relative Data

This kind of data are easy to find, since they can be found by load-register instructions, including LDR (unsigned word), LDRB (unsigned byte), LDRSB (signed byte), LDRH (unsigned halfword), LDRSH (signed halfword), LDRD (double word), with PC register being the base. PC register is not permitted to be an operand in all Thumb-2 instructions, and these instructions are examples that are permitted. Notice that PC value must be word-aligned, according to the document of ARM architecture [12].

3.2.1.2. Switch Table

Different compilers may use different patterns to generate switch tables. Since GCC [13] is open source, popular, and widely used in modern world, we can ensure that we can find all switch tables generated by GCC. There are three possible patterns that GCC generates for switch tables, as shown in Table 2. % indicates certain register, and # indicates an immediate value.

Table 2. Thumb-2 switch patterns

– cmp %case, #case_num – bhi #default_target		
tbb [PC, %case]	tbh [PC, %case, lsl #1]	adr %reg, #table_head ldr PC, [%reg, %case, lsl #2]

All Thumb-2 switch tables start with CMP (compare the value) and BHI (branch if greater than). First, the program compares the input value and total number of cases, and then jumps to the default target if the input value is out of range. The third instruction is decided by how many bits are used to indicate one case. TBB (H) uses one (two) bit(s) to indicated one target address, and the remainder uses four bits. The difference of them is what are stored in the table. The offset between target address and current PC is stored when TBB or TBH is chosen, while the target address is stored when the third pattern is used.

The third kind of switch pattern uses ADR instruction before LDR instruction and uses %reg instead of PC, because Thumb-2 has 16-bit and 32-bit instructions and the word data must be word-aligned. There might be a NOP instruction, regarding as a padding instruction, between LDR and the head of switch table if the address of LDR is not word-aligned. Since 16 bit ADR instruction ensure that the address stored in the register must be word-aligned, GCC generates ADR instruction before LDR instruction for fear that NOP influences the head address of switch table.

The offset between switch table and the branch target must be encoded in 9 bits and 17 bits for TBB case and TBH case respectively, and it must be positive (that is, only the addresses bigger than the table can be branch targets), so the third case is needed since the target address is stored immediately in the data, and all of the addresses in the binary can be

branch targets.

3.2.1.3. Padding

The purpose of using padding data is to make the instructions align word, half-word, or other length of bytes. NOP and NOP.w are frequently used for padding, and they can pad 16-bit and 32-bit respectively. Compilers also use 0x0000 as a 16-bit padding, and sometimes the padding size may be the multiple of 16 bits for some purpose. Fortunately, 0x0000 is regarded as MOVS r0, r0 when using Thumb-2 ISA; therefore, three cases of padding described above are all instructions, and they don't influence any CPU state and control flow of the program, so our translator doesn't have to regard these cases as special cases.

If other encoding methods of Thumb-2 instructions are also used for padding, they must be decoded and translated without any control flow reaching them. Even though they might consist of some undefined instruction, they can still be handled by regarding them as unidentified data, described in 3.2.1.4.

Our concern is the padding data that are not regarded as an instruction. This kind of data occurs when GCC generates switch table with TBB instruction and the total number of case is an odd number. Since Thumb-2 must be half-word aligned, a byte data must be padded.

As Figure 10 shows, there are nine cases, numbered from 0 to 8, in this switch table. The address 0xdf32 to 0xdf39 describe the target addresses of case 0 to case 7. Since we assume it is little-endian, the information of case 8 is "0x44". The next address to be decoded is at df3c, so there must be a padding byte at df3b (0x00 colored orange).

```
df2a:    cmp r3, #8
df2c:    bhi.n    df3c
df2e:    tbb   [pc, r3]
df32:    382c0738---------------------- 4 cases
df36:    38311450---------------------- 4 cases
df3a:    23000044---------------------- 1 case
```

Figure 10. An example of padding byte

There might be some strange cases that occurs in hand-writing code for generating padding data due to commercial issues. For example, pad a PC-relative data and some address of word is then marked as a data, so it won't be translated. In our experiment, PC-relative data are put at the end of functions, because the efficiency of pipeline may be influenced if they are put at the middle of a function; besides, there must be a branch instruction before these data. Therefore, the analyzer can determine whether the data address to load is really a PC-relative data by checking whether the instruction before this address is a branch instruction, or what before this address is also PC-relative data. The remaining case is that it might be the next function entry, and it can be determined by decoding from this data address and checking whether this instruction can be a function entry. The possible characteristic of function entries will be described in 3.3.2.1.

3.2.1.4. Unidentified cases

This kind of data occurs when LLVM disassembler returns fail state. Either these instructions are not supported by LLVM or they use some architectures, like vector float point (VFP), which is not normal architecture of ARM and Thumb-2. This kind of data may also be a kind of padding data. Our translator marks them as a kind of data and don't translate them for fear that some exceptions occur when executing our translator. The next word-boundary will be the next address for translating, because many Thumb-2 instructions require PC being word-alignment when executing; therefore, less mistranslations occur by handling in this way.

3.2.2. Code Location Problem

Since indirect branch targets can't be known by our translator in compiling time, a source-PC (SPC) to target-PC (TPC) mapping table is required for translated binaries, and TPC is the head address of an LLVM basic block in our translator. As a result, to make this table usable, our translator must maintain a mapping table from SPC to corresponding basic block.

This table is not difficult to generate, since required information is kept during translation. The difficulty is how to reduce the size of this table. The larger the table size is, the more time needed for optimizing and compiling; moreover, the execution time is also longer.

3.2.3. Other Problems

Occasionally, the pattern of certain kind of data may have a little difference compared with the normal pattern. For example, an instruction loads a word with the base register that stores the value of PC, and it must be a PC-relative load. Our system must have abilities to find this kind of situations.

Our translator translates all of the source binary into only one LLVM function, named “main”, so LLVM optimizer and LLVM static compiler may take too much time if the source binary is large since optimization unit is a LLVM function. For example, 445.gobmk in CINT2006, which has about 160 thousand instructions, takes about five hours for translation in our experiment. The complexity of them are at least quadratic, instead of linear, so our translator must have an ability to partition the main L-function into several smaller L-functions.

3.3. Implementation Detail

In this section, we describe how our translator solves issues described in previous section. A work we implemented may solve more than one issues, so we don’t name the title using the name of issue. The more information our translator gets, the more reliable the translated result is; therefore, we create an analyzer before calling the translator. Our analyzer glances the input binary first to get necessary information and pass them to our translator.

3.3.1. Find All Kinds of Data

3.3.1.1. Data Structure

Although how many memories needed when translating is no concern of our translator, we still want to use a better structure to record these information of the data position.

The most intuitive solution is that use several bits to indicate the data kind of certain half-word. This method is easy to implement, but it cost too much spaces. For example, a binary that contains about one megabytes may need about 200 kilobytes, about one fifth of

original binary. Besides, our analyzer and translator has to read every halfword one by one even if where it is translating is a large set of data. If our program can jump to the end of data once it attempts to translate the head of data, less time will be taken in translation.

If we define a part of continuous bytes with same kind as a set, then we can describe a set by giving its start address, end address, and its kind. Figure 11 shows an example. The input data in an undefined set, and our analyzer uses different kinds of methods to discover all kinds of data and marks them. These sets are mutual exclusive, so there is no ambiguous set after analyzing.

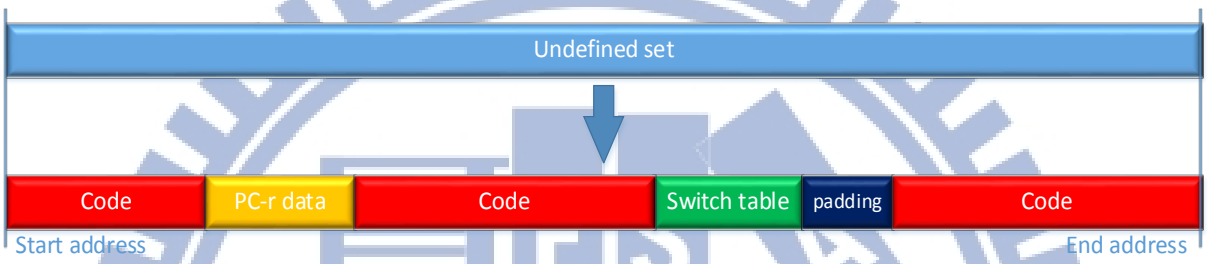


Figure 11. An example of set analyzing

Regarding continues bytes as a set is very useful, since it takes much less memories compared with using a bit map, especially when the data set is large, and it is also easy to tell our translator that it meets data and can skip. Furthermore, marking a little set in the undefined set is easy. Our analyzer just need to create an entry to store the start and end address of the set, and check whether there is the same kind of set before or after it. If true, these two sets can be merged, just like a union operation of two sets, as shown in Figure 12.

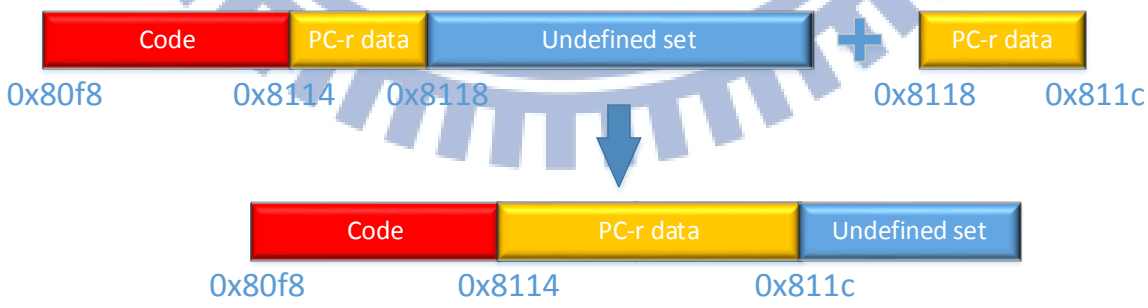


Figure 12. An example of sets union operation

Furthermore, this data structure has more benefits in saving memories. Since the bytes are read sequentially and their address sequence is strictly increasing, our analyzer can discard data whose address is smaller than current halfword. Take Figure 13 and Figure 14 as

an example, Figure 13 shows original version, that our analyzer don't discard any data that is used, and Figure 14 shows a discarding version. The green arrow indicates that where our analyzer is analyzing and the blue arrows indicate that where are LDR instructions that tell our analyzer where are PC-relative data. As a result, the discarding version uses less memory, since almost all cases of PC-relative data can be merged in only one set. In our experience, only two set entry is needed for the binaries generated by GCC.

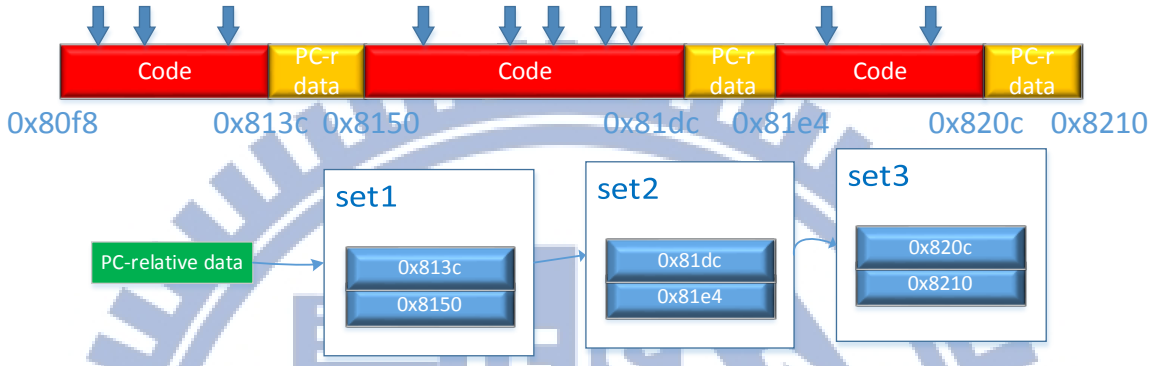


Figure 13. An example of non-discarding used data version

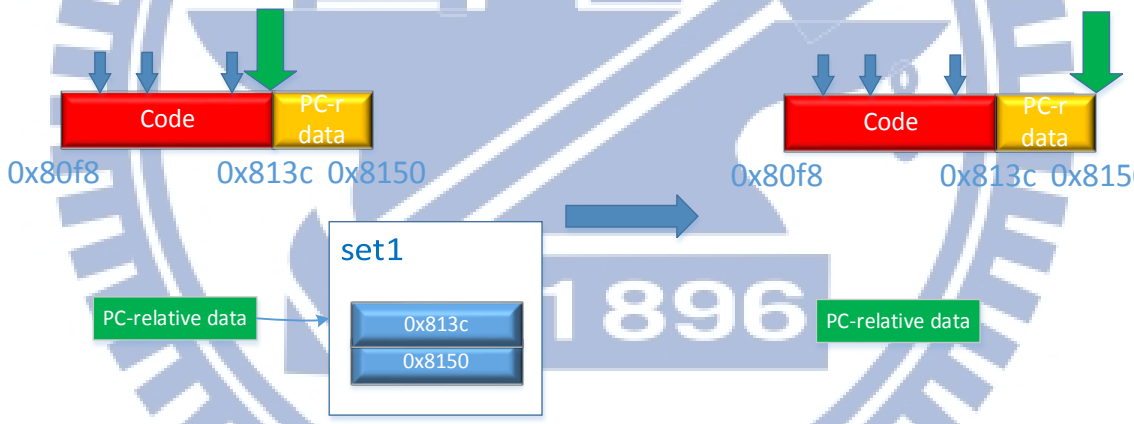


Figure 14. An example of discarding used data version

Our analyzer uses linked list to maintain different sets, as shown by Figure 15, since popping the front element and inserting the element at back are needed. Normally, a list of set is in increasing order, so our program just have to check the first set of the certain kind of list and decide what to do. Therefore, our program can execute faster.

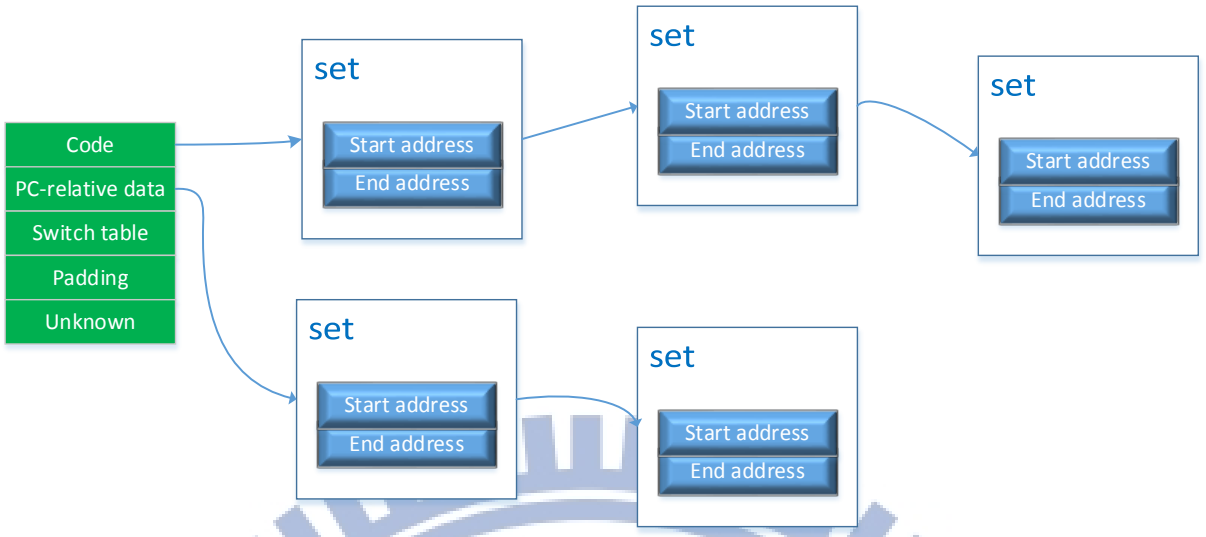


Figure 15. How these sets stored in the memory

3.3.1.2. PC-relative Data

Every time our analyzer finds LDR-prefixed instruction with base register, it passes associated information to the set handler. And this kind of data can be found correctly.

```
8120:    4b06          ldr   r3, [pc, #24]
...      ...           ...
8126:    4806          ldr   r0, [pc, #24]
...      ...           ...
813a:    bd10          pop  {r4, pc} (end of function)
813c:    00000000      .word 00000000
8140:    000bd36c      .word 000bd36c
```

Figure 16. An example of PC-relative data (using LDR)

Take Figure 16 as an example, at 0x8120, the address of the word program has to load is $0x8120 + 4 + 24 = 0x813c$, and $0x8120 + 4$ is the value of current PC. So our analyzer can mark 0x813c to 0x813f as PC-relative data. Alignment is also important in handling PC-relative data. At 0x8126 in Figure 16, $0x8126 + 4 + 24 = 0x8142$ is not the correct target address, because current PC is not word-aligned. The address must be $\text{Align}(0x8126 + 4, 4) + 24 = 0x8140$.

Due to the alignment of PC-relative data, our analyzer can ensure that the instruction at

the address which is not word-aligned is not start point of data, so the probability of mistranslating is lower.

3.3.1.3. Switch Table and Padding

Finite state machine (FSM) is used in our analyzer to find switch tables, because FSM is flexible and easy to implement.

The Left side of Figure 17 is our FSM for finding switch cases, and the arrows with no number indicate other cases that are not listed in the right side of Figure 17. Every time our analyzer reaches the final state, it receives necessary information about generating switch functions, like number of cases, default targets address and addresses of every case. The work of generating switch tables will be done by our translator.

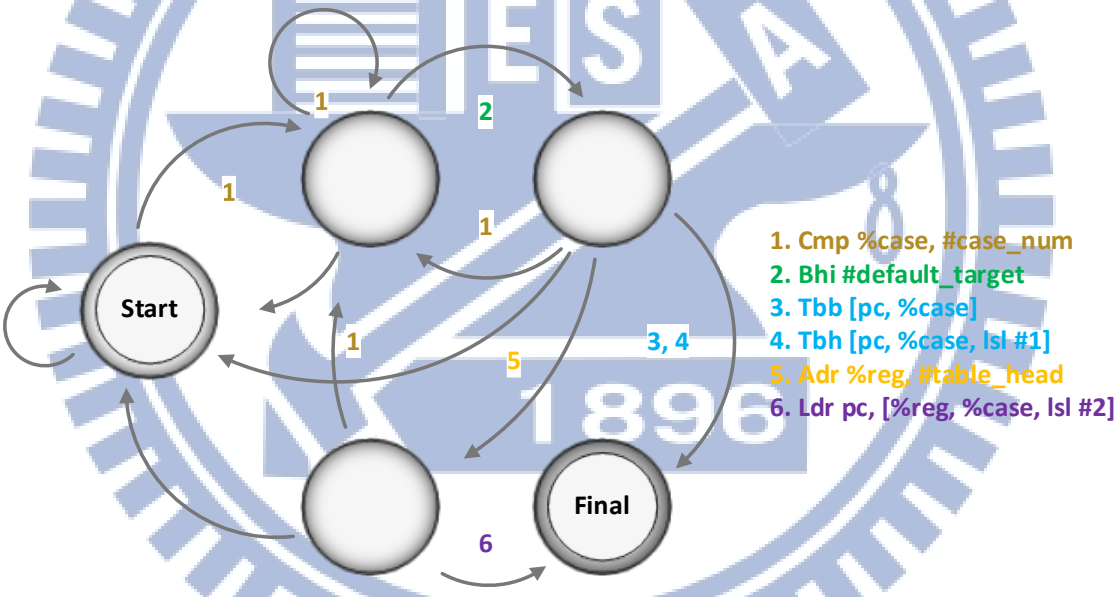


Figure 17. Finite State Machine for finding switch cases

If the input is TBB when entering the final state, our analyzer have to check whether the number of cases is an odd number, and mark the next byte after the table as a padding byte if true.

3.3.2. Address Mapping Table

Address mapping table (AMT) is created for finding indirect branch targets. The smaller the table is, the shorter the table looking time is. There are only two kinds of possible entries can be indirect branch targets: function entry and return addresses.

3.3.2.1. Function Entry

Function entries can be found in the symbol table, and this is the easiest case. Unfortunately, sometimes the symbol table is stripped, and our analyzer must have an ability to find the function entries.

There are several possible patterns can be considered as function entries; however, useless entries may also be regarded as function entries, so our analyzer must have ability to handle these entries. Besides, different cases may mark the same address as a function entry, our analyzer don't discard them since this information can be regarded as a profiling result. Since not all entries found in all cases are real function entries, our analyzer can regard these entries as function entries if it is hit more than certain times. In our experience, the patterns shown below can be a function entry.

1) The address of PUSH instruction with LR included.
2) The address of a 32-bits STMDB instruction with base register being SP and LR is included in operands. The behavior of this STMDB instruction is the same as PUSH instruction.
3) The next instruction immediately follows a POP instruction whose operands include PC.
4) The next instruction immediately follows a 32-bits LDMIA instruction with base register being SP and PC is included in operands. The behavior of this LDMIA instruction is the same as POP instruction.

```
000a0f2c <__GI_getpid>:
    a0f2c:      b543            push{r0, r1, r6, lr}
    …
    a0f3c:      bd4c            pop  {r2, r3, r6, pc}
    a0f3e:      bf00            nop
000a0f40 <__GI_gettimeofday>:
    a0f40:      b557            push{r0, r1, r2, r4, r6, lr}
    …
```

```
a0f68:    bd5e        pop  {r1, r2, r3, r4, r6, pc}
a0f6a:    bf00        nop
```

Figure 18. An example of PUSH and POP in finding function entries

5) The next instruction immediately follows a BX instruction with operand being LR.

```
000a59e0 <__pthread_mutex_lock>:
    a59e0:    2000        movs    r0, #0
    a59e2:    4770        bx    lr
000a59e4 <__pthread_mutex_init>:
    a59e4:    2000        movs    r0, #0
    a59e6:    4770        bx    lr
000a59e8 <_pthread_cleanup_push_defer>:
    a59e8:    6001        str    r1, [r0, #0]
    a59ea:    6042        str    r2, [r0, #4]
    a59ec:    4770        bx    lr
```

Figure 19. An example of BX in finding function entries

6) The branch target of BL and BLX instruction. Entries found in this case must be function entries, so our analyzer have to add all of them in the AMT.

7) The address of Function entries may be stored in the PC-relative data. Our analyzer has to check whether the data can be regarded as an entry, and adds it to the AMT if true.

8) The instructions that follow NOP or NOP.W instruction. Since NOP instructions are used for padding and make the address align certain bytes; therefore, this instruction has higher possibility to be put at the end of the B-function.

Sometimes there is a NOP instruction before the function entry because of the alignment issue, that is, our analyzer may regard the NOP instruction as a function entry in 3), 4) and 5). Our analyzer has to check whether this instruction exists for fear that adding no use entry in the AMT.

All of the B-functions end with branch instructions, no matter indirect or direct. Most of

the cases have been included as described above, but the cases that a B-function end with an unconditional branch instruction is not included.

Conditional branches can't be the end of B-function because the control flow must reach the next instruction if the condition fails, and instruction address following function call instructions are regarded as the return address, which describes in 3.3.2.2. As a result, only the address of instruction following B or B.W must be recorded. However, putting all of this kind of addresses in the AMT may cause the optimization time and compiler time much longer, even exhausting the memory. Therefore, our analyzer stores all of this kind of addresses in a secondary address mapping table, and lets the output binary search it if the searching of primary AMT fails. In our system, we maintain this secondary AMT by using LLVM switch instructions, which is the same as the original AMT and described in 3.3.2.3, and LLVM indirect branch instructions, which calls a helper function to search the target before it.

3.3.2.2. Function Return Address

Return address is the address that the function returns to. It is the address of the instruction that is immediately follows the function call instruction, like BL and BLX. Our analyzer stores all of this kind of entries in the AMT.

3.3.2.3. The AMT in LLVM IR

Our translator generate the LLVM switch instruction to handle the AMT, because it knows all of the entries from our analyzer, and using switch instruction is easier to generate direct jump to different entries. Since our analyzer only gets a portion of possible entries from the input binary, the AMT table will be too sparse if we put all of the entries in an LLVM switch instruction. Therefore, the compilers will generate a sequence of if-else instructions for switch instruction instead of a jump table. This will result in a bad performance for searching an address in the AMT. To solve this problem, we use a modulo-function as a hash function to split a large switch statement into several small switch statements.

Figure 20 shows an example of the AMT. The number of tables is dependent on how many possible entries our analyzer found, and it is assumed a power of two because simpler operation can be used when hashing. As a result, even the entry addresses in the level-2 table are still sparse, the number of if-else instructions must be much smaller than the one

without hashing.

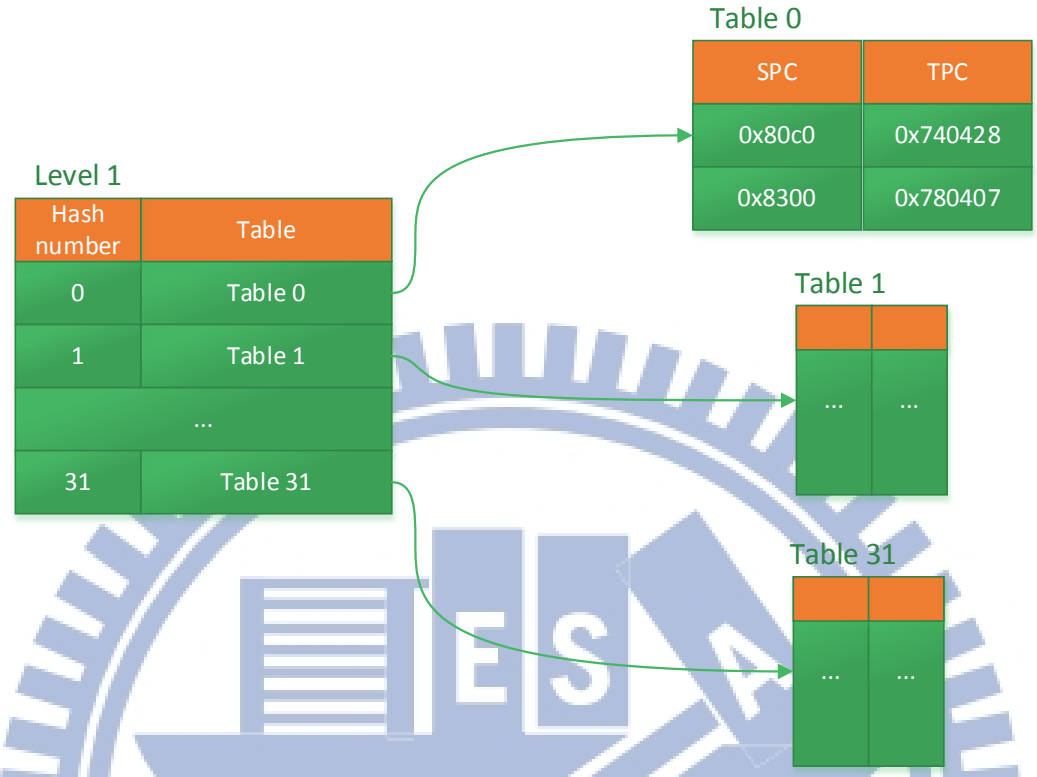


Figure 20. Diagram of the Address Mapping Table

3.3.3. Register value mapping table

In some cases, finding PC-relative data and switch tables is not as simple as what described above, due to some limitations of Thumb-2 architecture and different method used by the compiler.

For example, as shown in Figure 21, the address 0x8900 to 0x890b should be marked as a switch pattern by our analyzer, since the value of number of cases is stored in R2 at the address 0x88fa. This is because that the CMP instruction at 0x8900 is a 16-bit instruction, so only eight bits can be used for describing the immediate value. It is obvious that #558 can't be encoded using only eight bits. Another case is shown in Figure 22, the compiler generates an instruction that store data base, which is dependent on PC, in the register, and the following is an instruction that loads two words of data start at the base. This ADR instruction is put for alignment issue.

- 88fa: movw r2, #558

```
• 88fe:  subs r3, r0, #1
• 8900:  cmp r3, r2
• 8902:  bhi.w      9306
• 8906:  adr r1, 890c
• 8908:  ldr.w pc, [r1, r3, lsl #2]
• 890C:  ...(switch table)...
```

Figure 21. A special case of switch case pattern

```
• adr %reg_base, current_pc
• ldrd   %reg1, %reg2, [%reg_base]
```

Figure 22. An example of how GCC generates LDRD instructions

Two cases described above make our analyzer get wrong information that is important for our translator. Therefore, our analyzer should maintain a table that can record the value that is decidable in static time in the register, that is, what our analyzer records is the last value of each registers. Therefore, no matter how many instructions are located between the "get value" instruction and "use value" instruction, our analyzer definitely gets the correct information it needs. Moreover, this table should be flushed when reaching branch instructions for fear that the analyzer gets wrong information.

Constant folding and constant propagation technology can also be implemented using this table. However, our translator is static and the LLVM optimizer has ability to take these optimization, so we did not implement these optimization here.

3.3.4. Partition the main L-function into several L-functions

The complexity of LLVM optimizer and static compiler are at least polynomial with power greater than one. Since our translator works quickly, the speed of LLVM optimizer and static compiler dominates the execution time of our system. If we can lower the size of the main function, our system can generate translated binary faster.

The comparison between one function method and multi-function method, which distributes instruction blocks uniformly, are shown in Figure 23. Furthermore, the distribution method can be changed if the user wants. We introduce how our analyzer and

translator partition the main L-function and distribute them into different L-functions. Moreover, we also introduce the method we implemented for handling function switching and some modification for fitting the optimizations of LLVM optimizer.

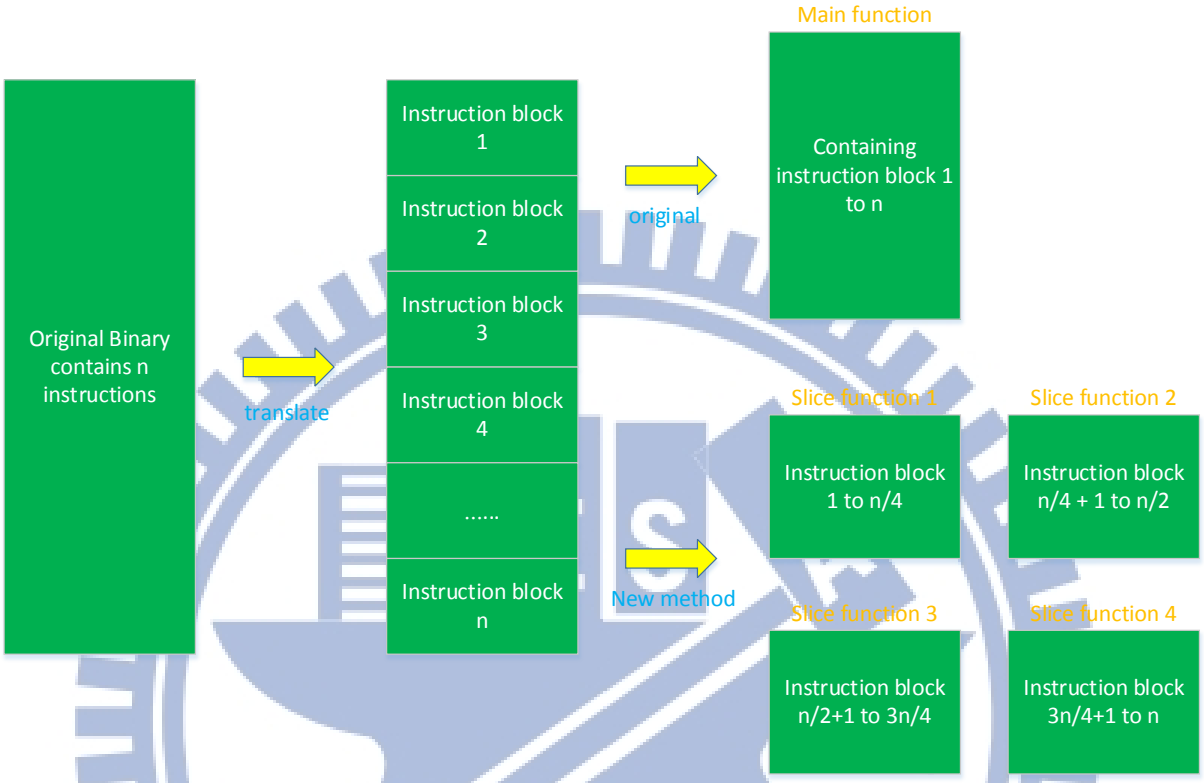


Figure 23. Comparison between one function and multi-function

3.3.4.1. Overview

Figure 24 is the framework of the LLVM module that our system generates using multiple LLVM functions. Since the architecture states and Application Program Status Register (APSR) are regarded as global variables in the LLVM module, we don't have to pass any of them as parameters when switching L-functions.

In multi-L-function version, the main L-function just handle some initialing routine, like initialing stack space, putting command line arguments and environmental setting in the stack, and then jump to the L-function that contains the instruction block whose address is entry point of the input binary; moreover, since frequently switching between functions may cause stack overflow, we let the slices of L-function return to the main L-function if needed. Therefore, the main L-function must have an ability for handling function switching, regarding as a switching stub.

Other slices of L-function may call others if branch target is not in it. Besides, indirect branch table is also partitioned in several part, and they are appended to the end of each slice of L-function.

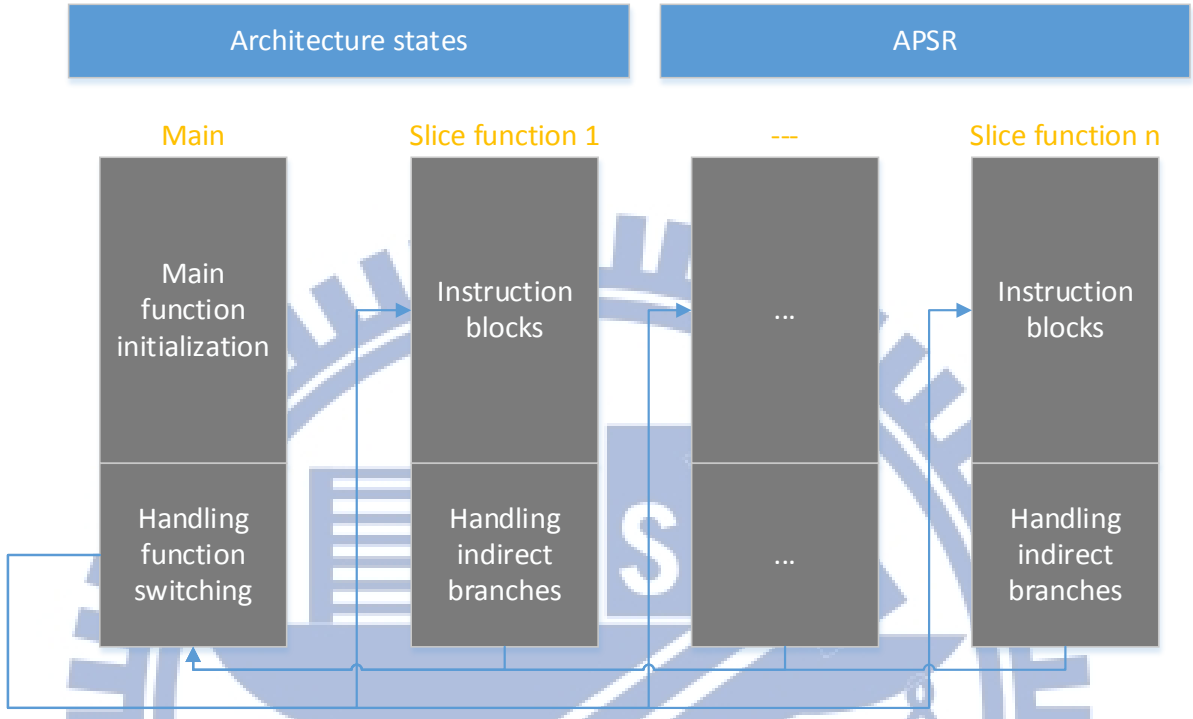


Figure 24. The framework of multi-function version LLVM module

3.3.4.2. How many LLVM functions have to be generated

The size of each L-function is user defined and the size of all instructions in the input binary divides by this user defined value; however, we have to avoid that certain L-function becomes too small, so L-function size is permitted to be a little larger than defined if this situation happens.

3.3.4.3. Strategy for distributing instruction blocks

The cost of function switching is not high, but this cost may influence the performance seriously if they switch frequently. Our goal is to lower the frequency of function switching by finding instructions that have more probability to be executed sequentially. Therefore, the instructions in the same B-function must be distributed to the same L-function.

Since the only useful information our analyzer can get is the target address of calling

function, so our analyzer has to find all function entries in the input binary first. How to find the entries of B-function has been described in 3.3.2.1. Then our analyzer constructs a graph, regarding each function as a node and a calling action as an edge. Take Figure 25 as an example, F1 calls F2 and F3, so there are one direct edge from F1 to F2 and one from F1 to F3.

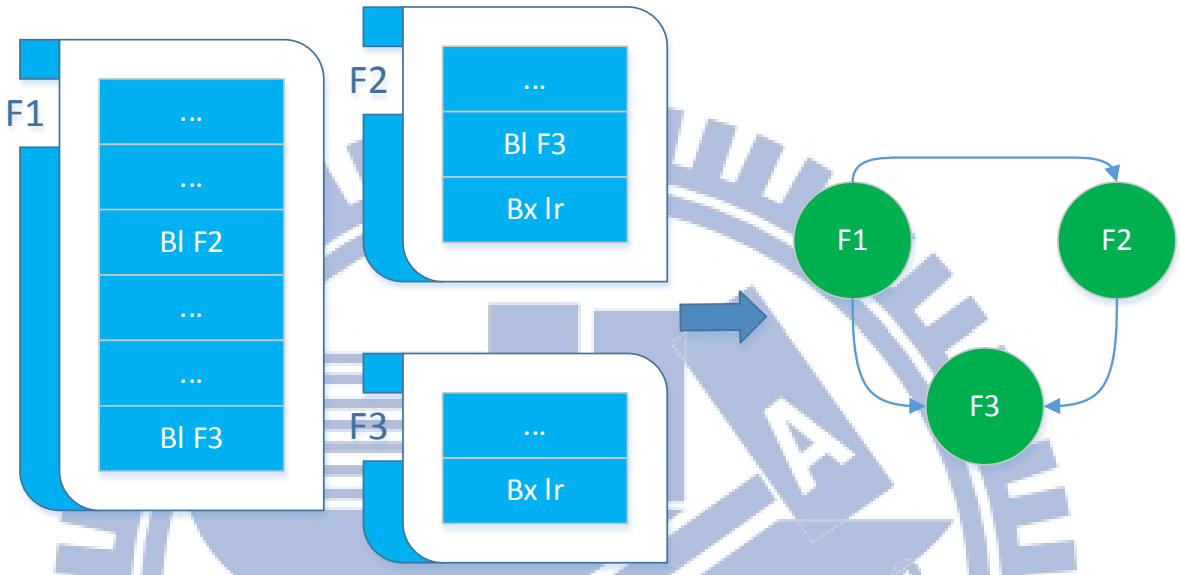


Figure 25. An example of function graph

After generating this graph, our analyzer performs depth first search (DFS) to find all connective components and put the B-function pointer in the list. Each search is start at the node with zero in-edge, that is, no other B-function calls it. However, since the graph is directional, this approach will find two components if two B-functions which no other B-function calls them calls the same B-function. To have more probability that the same component are distributed to the same L-function, our analyzer regards the case described above as only one component.

Finally, our analyzer sorts all components according to their size, and then distributes them, beginning with the largest one. A component will be split into several components if no L-function has enough space to hold it. Besides, some flexible coefficients are added here for fear that some component is just less than 10% larger than the maximal space. This approach may lower the compiling speed, but it is worth since the component won't be split.

We choose DFS instead of other searching method like breath first search (BFS), because the recording order of B-functions in a connective component is based on which is traced

first. We should ensure that not only the B-functions in the same connective components but also the longest path in the graph has more probability to be distributed into the same L-function.

Our analyzer also implements uniform distribution method. It just distribute B-functions sequentially until the size of L-function larger than user defined threshold. Although the performance of this method is usually worse, it has advantage when handling function switching, as described in the following section.

3.3.4.4. Function mapping table

To find where the instruction block of indirect branch target is, our analyzer has to maintain a table for this purpose. Since indirect branch targets are either function entries or return addresses, we can use the same method used in 3.3.2.3, except that it uses call-instruction instead of branch-instruction.

We can also use helper function to find the target LLVM function. Binary search is used in this helper function, and each entry contains address of B-function head, address of next B-function head, and L-function it was distributed. The helper function claim that it finds the target if the target address is between the first two values of the entry. The advantage of this approach is that if the target is not one of the function entries, the translated program can still find the corresponding LLVM function. Although binary search is the best search algorithm, it still takes time if it searches the same target frequently. Therefore, we also add a function table cache to enhance the performance.

Using helper function is efficient if uniform distribution strategy is used. Since the B-functions that are distributed to the same L-function are put continuously in the input binary, these functions can be described in the same table entry. As a result, the number of entries used in helper function is equal to the number of slices of LLVM functions, and the search time becomes very low.

3.3.4.5. Function switching

The easiest method for L-function switching is just call another L-function directly, but this approach occasions stack overflow if switching behavior occurs frequently. We use two

kinds of method to avoid stack overflow occurs. The first one is returning to the main L-function and then call another function, and the second one is returning to the main L-function when the counter is larger than the user-defined threshold.

Furthermore, the main L-function should have ability to handle function switching, so the target address should be passed between the main L-function and other slices of L-function. Therefore, we declare that all slices of L-function have two parameters, the target address and the counter for calling (omitted if the user sets that the function returns to the main every time before function switching), and two return values, the function to call next time and the target address. If the first return value is zero, then the program will look up the function mapping table and then call the corresponding L-function. Figure 26 is an example that uses switch cases to implement the function mapping table, and its input is the return value of the previous L-function.

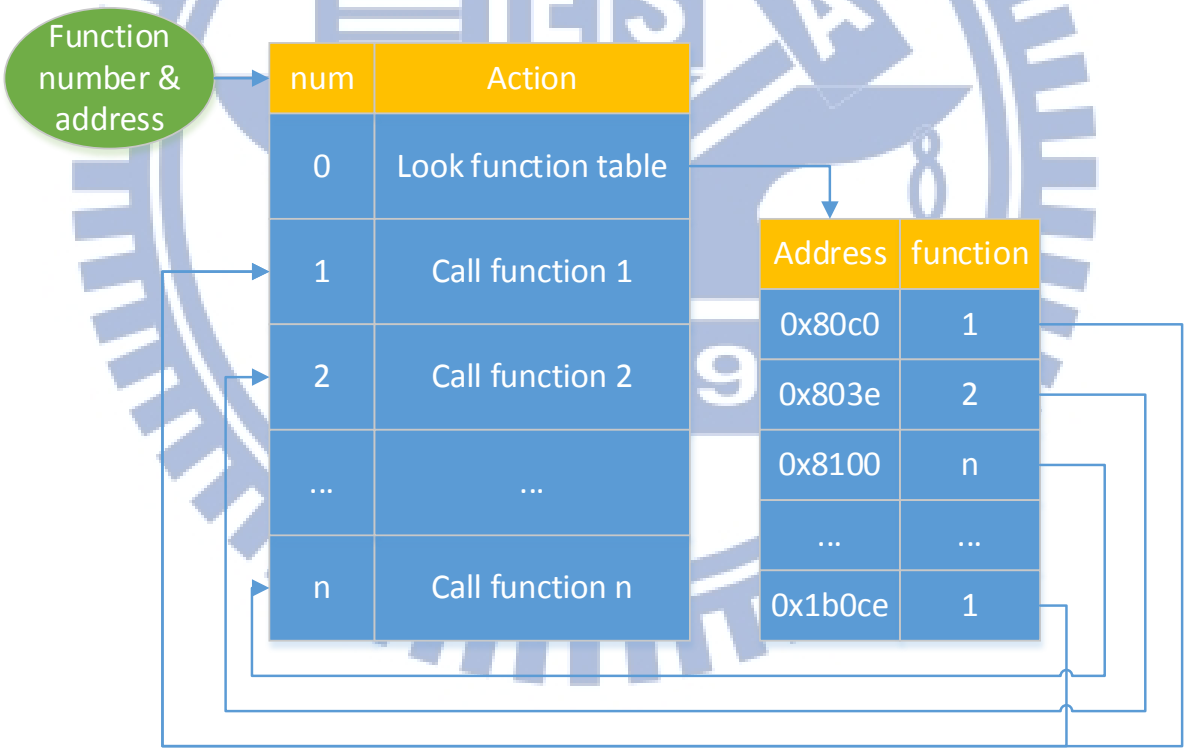


Figure 26. An example of function switching handler

The last modification is that we have to let all slices of L-functions jump to the correct instruction block according to its first parameter, the target address. Since function switch always occurs when calling function or returning to the caller, which is the same as the condition that indirect branches occur; therefore, the same looking table can be used for finding the target. We add a branch instruction at the head of slice of L-function and then it

can execute at correct position. Figure 27 illustrates the control flow of each slice of L-function. The arrow A and C indicate what we explained above; besides, B, C and D indicate the flow when handling indirect branches. The mapping table is designed for two purposes, so the code size of the binary generated by our system can be smaller.

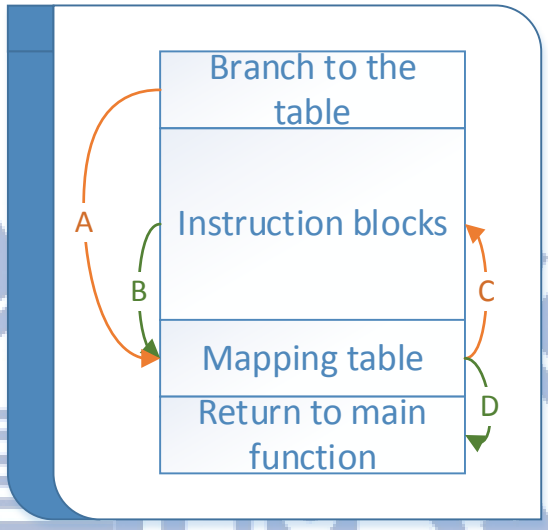


Figure 27. The control flow of each slice of LLVM function

3.3.4.6. Local variable remapping

In some instructions, the result is stored in the local variables, and these local variables are created for the main L-function. Therefore, these variables become undefined after partitioning the main L-function in to several slices of L-function. We should create these variables for every slices of L-function and substitute the operand of these instructions by the new ones. For convenient, our translator marks instruction blocks that contain several uses of local variables, so this substitution can be performed more efficiently.

3.3.4.7. Global variables remapping

The architecture states and the application program status registers are regarded as global variables in our system, and they always accessed by memory load and store operations. Fortunately, LLVM optimizer provide a memory-to-register optimization that substitutes these memory operations by mapping to certain registers, and the performance enhances tremendously. This optimization find all of the allocation instructions in the bitcode file, and promotes the local variables to registers. An example is shown in Figure 28, and only one memory reference is needed after optimizing.

%0 = load i32* @SP %1 = add i32 %1, -4 %2 = load i32* @R0 %3 = inttoptr i32 %1 to i32* Store i32 %2, i32* %3 Br label %L_next	%0 = add i32 %SP.0, -4 %1 = inttoptr i32 %0 to i32* Store i32 %R0.0, i32* %1 Br label %L_next

Figure 28. An example of mem2reg optimization (STR r0, [SP, #-4])

The memory-to-register optimization is only available on local variables, but our system regards architecture states and APSR as global variables. The reason why the original version that uses the main L-function only can also optimized with memory-to-register optimization is that the LLVM optimizer performs a global variable optimization that converts global variables to local variables if the global variables is only used in one LLVM function.

According to the reason described above, what our system has to do is substituting global variables by local variables, and let the global variables only appear in the main function. However, the latter goal is discarded since all global variable must be passed to other LLVM functions and the only way is regard them as parameters; therefore, about twenty parameters make reliability of our system lower because of the rapid growing of stack. Since global variables are not accessed frequently in the main L-function, the performance won't be influenced too much without the memory-to-register optimization in the main L-function. As a result, the only modification our system has to do is loading all global variables in the local variables when entering the LLVM functions, and storing the latest value of local variables back to the global variables.

3.4. Relaxing the Restrictions

The limitations of our system are shown in 3.1, and our system can be more powerful if some of them can be relaxed or removed. We just discuss how to relax the restriction that the binary must be generated by GCC, since this is the largest obstacle for us when claiming that we have solved the code discovery problem in Thumb-2 instruction set. Other restriction may also be relaxed, and we regard them as future works.

3.4.1. Using the compiler other than GCC

GCC is one of the most powerful and popular compiler in the world, and it is also open-source, so we can easily know how to find the data, including PC-relative data and switch tables, in the GCC-generated binary by tracing source code of GCC. For other compilers, we have to separate them into two categories according to whether it is open-source.

PC-relative data can usually be found by searching LDR-prefixed instructions with base register being PC. However, some compilers generate this kind of instructions by storing PC to certain register and using this register as the base when loading. Fortunately, we have implemented the register mapping table as described in 3.3.3, so these cases are easy to be found, and we can add some routine to our analyzer easily if the pattern can't be recognized. Therefore, no matter whether the compiler is open-source, PC-relative data can be easily detected.

Finding switch tables is more complex. If the compiler is open-source, then it's easy since a new finite state machine can be added to our analyzer and the corresponding patterns can be recognized. However, the popular compilers for ARM, like ARMCC and Microsoft ARM compiler, are not open-source; therefore, what we can do is using many test cases to try what they generate for switch tables and then we can create a FSM corresponding to them. This approach is dangerous since we can't ensure our analyzer can recognize all of the patterns that the compiler may generate. As a result, we have to do some analysis about the compositions of switch tables, and this work is shown in next section.

3.4.2. Switch Table Analysis

The switch table must contain several factors, including table base, default target, number of cases, and the value that describes the jump offset. Once our analyzer finds them, it knows where the table is, so we will discuss how to find these values in the binary.

Thumb-2 instruction set has two instructions, TBB and TBH, which implement table branch; therefore, compilers that use only these instructions to generate switch table is reasonable since hardware implementation must be better than software. As a result, we analyze the use of TBB and TBH first. Encoding method of TBB (TBH) are shown in Figure 29, it loads a byte (halfword) from the address which is the sum of <Rn> and <Rm> (<Rn> and

<Rm> * 2) and branch to the address whose value is current PC plus the loaded value.

Encoding T1 ARMv6T2, ARMv7

TBB<c> [<Rn>, <Rm>] Outside or last in IT block

TBH<c> [<Rn>, <Rm>, LSL #1] Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn				(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	H	Rm			

Figure 29. Encoding method of TBB and TBH

First, we have to claim that the TBB (TBH) must be decoded correctly. If the table is put after TBB (TBH), then the instructions before TBB (TBH) must be decoded correctly and so as TBB (TBH). If the table is put before TBB (TBH), there must be a branch instruction before the table since this table is generated for the instruction after it. As a result, we can claim that the TBB (TBH) for the switch table is definitely be found.

<Rn> of TBB (TBH) is the table base and <Rm> is the offset, so what our analyzer have to find is default target and number of cases, now. The offset must between zero and the number of cases; otherwise, the table basis must be other value. Therefore, there must be a conditional branch instruction that branches to the default target, and a comparing instruction that decides whether the branch is taken, and they must be found before TBB (TBH).

Testing whether a value is between zero and another value takes two instructions, so the compiler may shift all of the case number and let the smallest one be zero. Therefore, only one comparing instruction is needed, and it tests whether the value is less than certain value. Furthermore, this upper bound can be regarded as the number of cases. What we have to notice is that the table size must be a multiple of two due to the alignment issue, so the number of cases must be plus one if it is an odd number. In general, this comparing instruction is CMP, which uses subtracting operation and updates APSR without saving result, and the following with the instruction B, with the condition being greater than or equal, just greater than is also permitted. As a result, if our analyzer can find a comparing instruction and a branch instruction in ordered before TBB (TBH), then the corresponding switch table can be found.

Compilers may generate switch table using instructions other than TBB and TBH, but the

comparing instruction and branch instruction are still necessary. Since the instructions that can be used to load a data and jump is not many, we can add all of them in our analyzer that fits all of the cases.

3.4.3. Case study: ARMCC

The ARM compiler (ARMCC) usually uses LDR-prefixed instructions to load PC-relative data, and it also uses ADR (form a PC relative address) instruction to store PC value in the register and then load the data by LDM (Load multiple) instruction.

In some cases, PC-relative data generated by ARMCC may be null-terminated. For example, the only parameter passed to the “printf” function is only the head address of the data. This case don’t occur in the binary generated by GCC since GCC puts null-terminated data in other section. Our analyzer don’t support this kind of PC-relative data, since we don’t have enough information about how many functions use this kinds of data.

For switch tables, ARMCC uses CMP and BCS (branch greater than or equal) and following TBB or TBH to implement switch tables, and what the difference between it and GCC is that GCC uses BHI (branch greater than). Therefore, the difference is that the number of cases in ARMCC that is gotten from CMP instruction have to plus one. Some cases may not be found by our test cases since ARMCC is not open-source, but we have shown that our analyzer can cover almost all of the cases with a few modification.

The system call instruction generated by ARMCC use ARM semihosting interface, instead of using Linux kernel, so our system call emulator doesn’t support the binaries generated by ARMCC now. We tried to translate the binary generated by ARMCC just because we want to ensure that what we did for code discovery problem can fit other compilers easily.

IV. Experimental Results

In this chapter, the performance, including execution time, and translation time, and the code size of the binaries generated by our SBT system are shown. Some analysis about our results will also be described. Since LLVM IR cannot be executed directly, we let our system generate x86 executable using LLVM infrastructure and then compare the results that execute on x86 machine.

4.1. Environment

We use EEMBC version 1.1 [19] and SPEC 2006 CINT as our benchmark. The Thumb-2 binaries, that is, source binaries, of these benchmarks were compiled using GNU GCC 4.4.6 with –O2 optimization. Moreover, in order to reduce the size of the statically linked binary and its translation time, µClibc [20] with the version 0.9.32 is used when linking the object files. The LLVM version used in our SBT system is 3.2, which is the newest version when we finished developing our system. The LLVM optimizer uses default optimization setting, that is, -O2, to optimize the target binaries translated by our SBT. The option "-mem2reg" which performs memory to register mapping optimization is also selected, since all of the architecture states and application program status registers are stored in the memory, and they can only be accessed using load and store operation. Since memory references takes a lot of time, this optimization will lead tremendous enhancement. There might be other optimizations that can improve the performance of translated binaries, like –early-cse, -reassociate, -gvn, and –instcombine [21], but how to choose the best combination of optimization options is not the purpose of this thesis; therefore, we just chose –mem2reg optimization. All of the benchmarks run on Intel Core i5-3470(3.2GHz) with Ubuntu 13.04 32-bit operating system and Linux kernel 3.8.0. The version of QEMU we used is 1.4.0.

4.2. Performance

In this section, we compare the performance between the binaries generated by our SBT system, the binaries generated by native system and the binaries translated by QEMU,

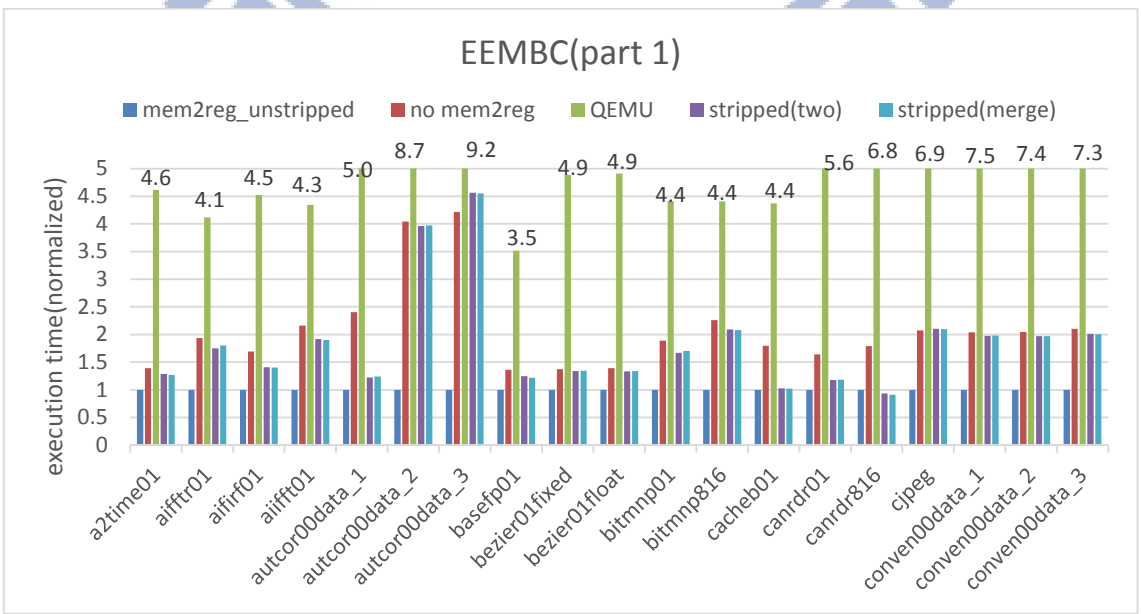
which is a popular DBT. The reason why we chose QEMU is that it is a popular binary translator and is reliable. The results of some benchmarks are likely incorrect compared with the result using native system, so we use QEMU to ensure that the reason that makes the result incorrectly is not in our system. We will show the results with different parameters. The comparison about translation time, and some statistic results will also be shown.

4.2.1. Execution Time

If there are still some kinds of data that is not found by our system in Thumb-2 binaries, the result of executing translated binary will also be incorrect. Therefore, what influence the performance of the translated binaries are the method for handling address mapping table and the options being given to LLVM optimizer and LLVM static compiler. How to handle indirect branches has been described in 3.3.2.

4.2.1.1. EEMBC

The results of execution time of EEMBC are shown in Figure 30. We normalize the results by the fastest one, which uses –mem2reg flag when optimizing. The inputs of the front three cases are unstripped binaries, while stripped binaries in last two cases. The difference of the last two cases is that they put the address after unconditional branches in the secondary address mapping table and the primary mapping table respectively, and the reason for adding such entries has been described in the last paragraph of 3.3.2.1. Besides, the last two stripped cases are also use –mem2reg flag.



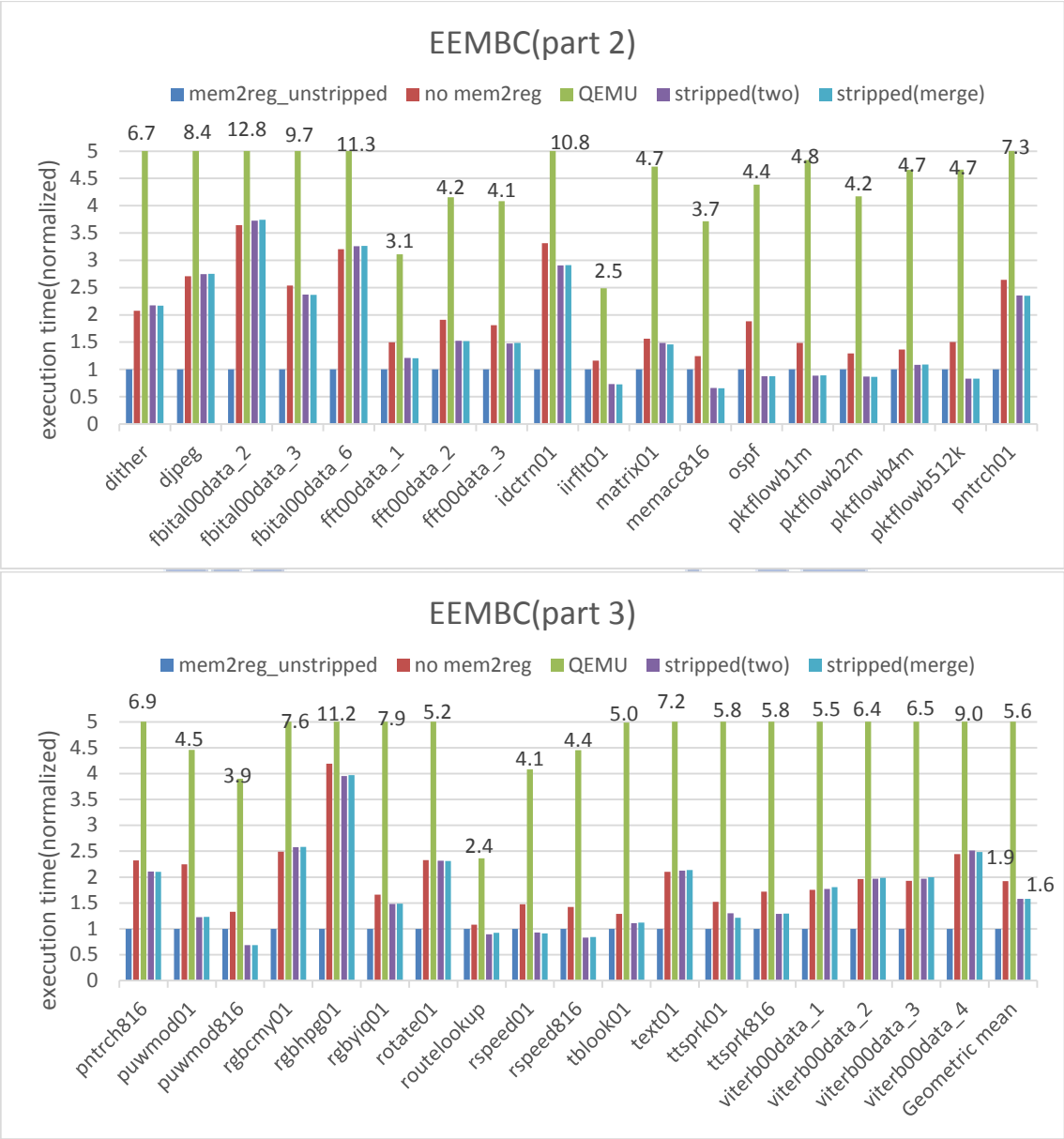


Figure 30. EEMBC execution time

For all of the EEMBC benchmarks, QEMU is much slower than others. This is not a surprise for us since SBT must be faster than DBT because SBT spends time for translation in static time, which is off-line and not included in the execution time. We can take a look at the geometric mean of all cases, which is the last item of part 3 of Figure 30. The –mem2reg flag provide about 1.9 times of performance enhancement, with the smallest address mapping table; therefore, we always choose this flag to get better results. No matter how the entries after unconditional branch are accessed in the address mapping table, both of two stripped cases spend about 58% more time, which is very slow. The reason is that the code size of EEMBC benchmarks are small, only about ten thousand instructions; therefore, the corresponding address mapping tables are also small, about 800 entries for unstripped input

and 1200 entries for stripped one. Since the number of return address is a constant in both AMTs, the quotient of the size of stripped and unstripped AMT becomes higher if the number of return address is small. In EEMBC, the number of return address is about 600 in average, which is 73% of the size of unstripped AMT. We will show that the return addresses dominate the size of the address mapping table in common-use program by some statistical result of CINT2006. As a result, the higher quotient of the size of two kinds of AMT make the performance be lower. Another possible reason is that the very short execution time, which is shorter than one second, make the ratio of the time spent in searching AMT and the total execution time become higher, so the difference of the execution time is magnified.

We list some statistical results in Table 3, and the last column is the sum of previous two columns, indicating the entries in the case which puts the address after unconditional branches into the primary address mapping table. The column named “Total” is the total instructions in the binary and the column named “AMT” is the size of address mapping table when using unstripped binaries. In the figures above, we can see how this approximately two times larger address mapping table influences the performance.

Table 3. Statistical information about some of EEMBC benchmarks

Set	benchmark	Total	AMT	AMT for stripped	Secondary	Sum
automotive	a2time01	10135	762	892	374	1266
	basefp01	9726	772	884	367	1251
consumer	cjpeg	17016	1364	1573	593	2166
	djpeg	18357	1355	1582	664	2246
networking	pktflowb1m	9778	770	903	379	1282
	routelookup	9528	747	854	379	1233
office	bezier01fixed	9382	735	862	357	1219
	bezier01float	9382	735	862	357	1219
telecom	autcor00data_1	10200	865	1006	390	1396
	conven00data_1	10111	855	990	387	1377
Average of all (about 60 bmks)		10425	819	950	393	1343

In some cases in above figures, stripped cases are better, like canrdr816, memacc816 and puwmod816, which is unreasonable. Since the execution time of these benchmark are very short, less than 0.01 seconds in our experiment, indirect branches may seldom occurs and some table entries may be discarded by LLVM optimizer. Besides, system overhead may

also influences the performance, especially when the execution time is very short, so we test our system using different benchmarks many times and take the average.

4.2.1.2. SPEC 2006 CINT

We choose both test data and reference data in SPEC 2006 CINT to test our translated binaries. There are twelve benchmarks in CINT2006, but some of them are not runnable or generate incorrect result, the summary is shown in Table 4. Some problem, like memory issues in 400 and 473, are what mc2llvm should solve. Since the objective of this thesis is not finding a better way to map the memory layout from source binary to target binary, we remain these two benchmark being not runnable. System call “fork” is regarded as a future work of our system, because multi-thread backend must be supported for this purpose. The remaining problem are not what we can control, since the compiler for only Thumb-2 instruction set is not popular; therefore, more limitations are added in this compiler, like some features must being disabled. As a result, ARM/Thumb-2 mixed ISA is the most probably the next ISA that our system supports.

Table 4. The reasons for not runnable benchmarks in CINT2006

	Test data	Reference data
400.perlbench	Our system doesn't support fork system call, which exist in one of the test cases.	Heap reaches 0x08048000, which stores read-only data when executing an executable.
403.GCC	Instruction size is too large to be compiled, and internal compiler error occurs due to some flag setting issues.	
464.h264	Floating point issue, which makes the result incorrect. Since the result of QEMU is also wrong, it should be a cross-compiler setting issue.	
473.astar	Runnable	Heap reaches 0x08048000, which stores read-only data when executing an executable.
483.xalanc	Can't be compiled by our cross compiler due to some setting issues.	

The results of execution time of CINT2006 are shown below: Figure 31 show the result of test data while Figure 32 show the result of reference data. We use the execution time of the binaries compiled directly for our native system (with GCC 4.4.6 and –O2 optimization option) as the base, and compare it with the others. The third and the fourth column list the result without –mem2reg flag and with –mem2reg flag when optimizing the translated

bitcodes respectively. Both of stripped columns put the address after unconditional branches, which can be a function entry, as described in 3.3.2.1, in the secondary address mapping table, and the first one accesses the table by LLVM switch instructions while the second one uses LLVM indirect branch instructions with a helper function for searching the entries by the binary-search algorithm. The previous one always performs better, because more optimizations can be applied by LLVM optimizer. However, when the size of possible entries grows, LLVM optimizer and LLVM static compiler may have no ability to scale this complex work, because they cost too much memory when optimizing. As a result, some translated bitcodes can't be optimized with aggressive optimizations, like 471.omnetpp. The result of 471 with the first stripped version remains blank because it costs too much memory when compiling to x86 assembly.

(measured in seconds)	Native	QEMU	no mem2reg	mem2reg unstripped	Stripped (switch)	Stripped (helper)
401.bzip2	5.46	53	18	6.66	7.55	8.21
429.mcf	1.72	7.27	3.01	1.86	1.97	2.02
445.gobmk	15	188	70.8	36.9	37.1	37.2
456.hmmer	3.84	110	34.1	20.9	20.9	21.4
458.sjeng	3.35	46.5	14.8	7.04	7.77	8.1
462.libquantum	0.0408	0.762	0.196	0.0752	0.0825	0.0854
471.omnetpp	0.328	10.6	3.82	3.32		4.17
473.astar	8.38	51.4	17.6	11.9	12.4	12.9

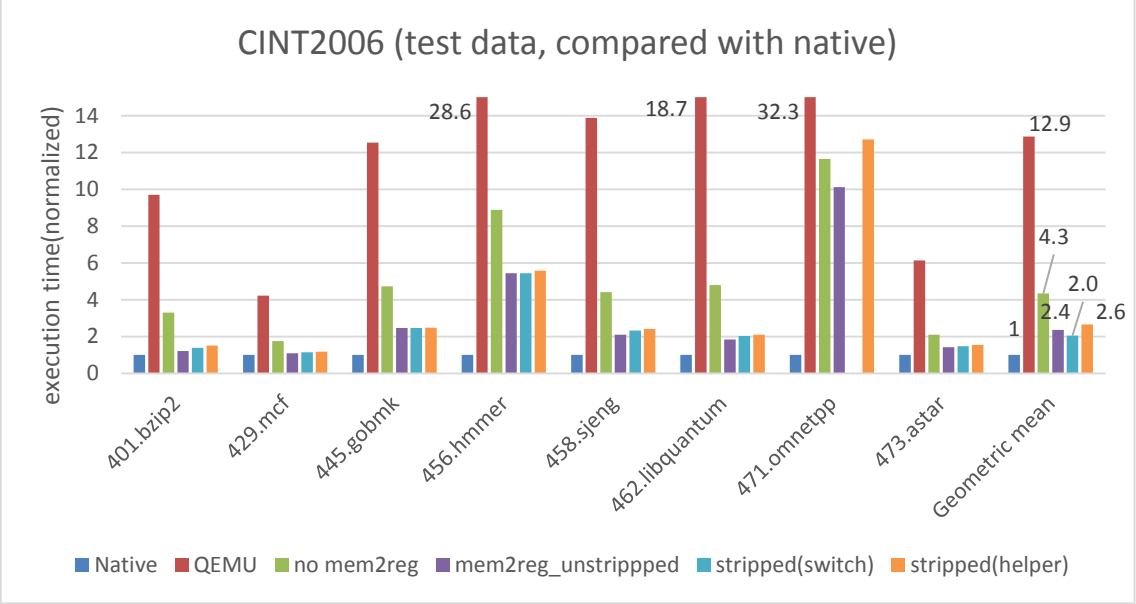


Figure 31. Result of CINT2006 with test data, compared with native result

(measured in seconds)	Native	QEMU	no mem2reg	mem2reg	Stripped (switch)	Stripped (helper)
401.bzip2	482	4670	1538	621	716	756
429.mcf	245	854	457	260	269	275
445.gobmk	412	5676	2093	1095	1106	1108
456.hmmer	409	8313	2942	1064	1052	1089
458.sjeng	483	7188	2272	1065	1175	1238
462.libquantum	347	4585	1170	370	450	426
471.omnetpp	272	4518	1293	1266		1445

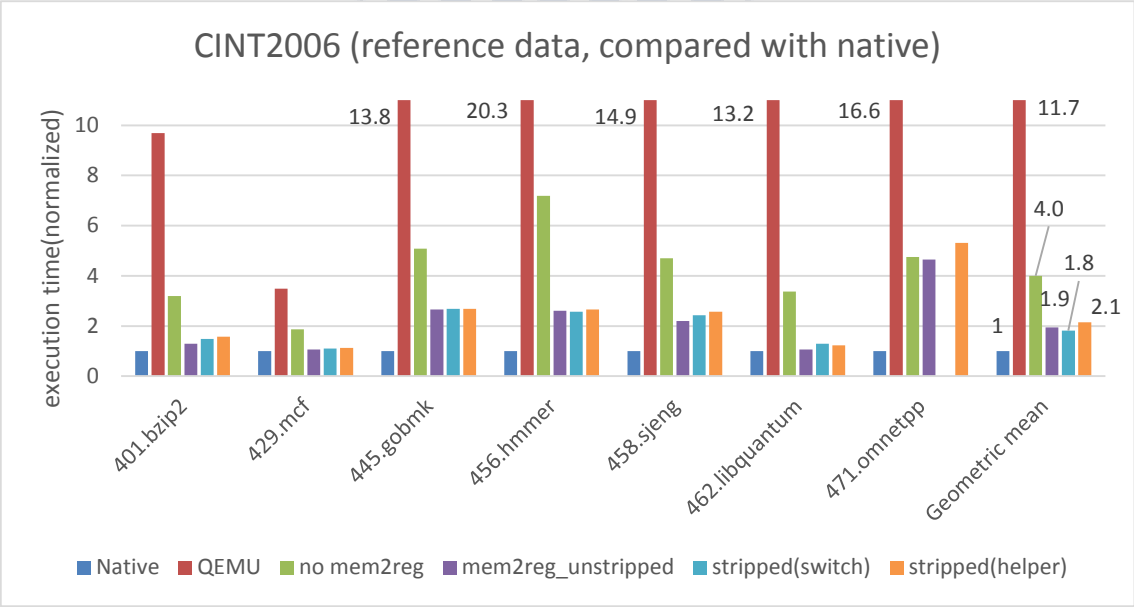


Figure 32. Result of CINT2006 with ref data, compared with native result

For test data, our best result takes 2.4 times more time than the native system in average and 1.9 times more for reference data; however, the performance of 456 and 471 are very bad compared with native system. Since the compiler may generate instructions with better performance when directly compiling, we think it can't be better without other modification when translating to LLVM IR or adding other optimization option when optimizing. By the way, the performance of the first stripped version is better than our best result, but it omits the result of 471, which other cases perform terrible result; therefore, we can't conclude anything about this smaller value.

Take a look at Figure 33 and Figure 34, which is normalized by our best case, for more detail. We omit the result of QEMU here since its performance is not good, even though many parts of translated code must have been in the code cache of QEMU after long time execution. In average, the execution time of QEMU is 5.4 and 6.1 times more than our best

result, with test data and reference data respectively. Besides, both of stripped cases are only about 10% slower, which gives the credit to the good method for selecting possible function entries, described in 3.3.2.1. The performance difference of them are not large, but the difference of the translation time, including translating to LLVM IR by our translator, optimizing by LLVM optimizer and compiling by LLVM static compiler, are large, as shown in Figure 35. The reason in that LLVM switch instructions take more time to be optimized. If how much time is spent in translation time is not important and the input program is not too large, using the first solution, which uses LLVM switch instructions, is recommended.

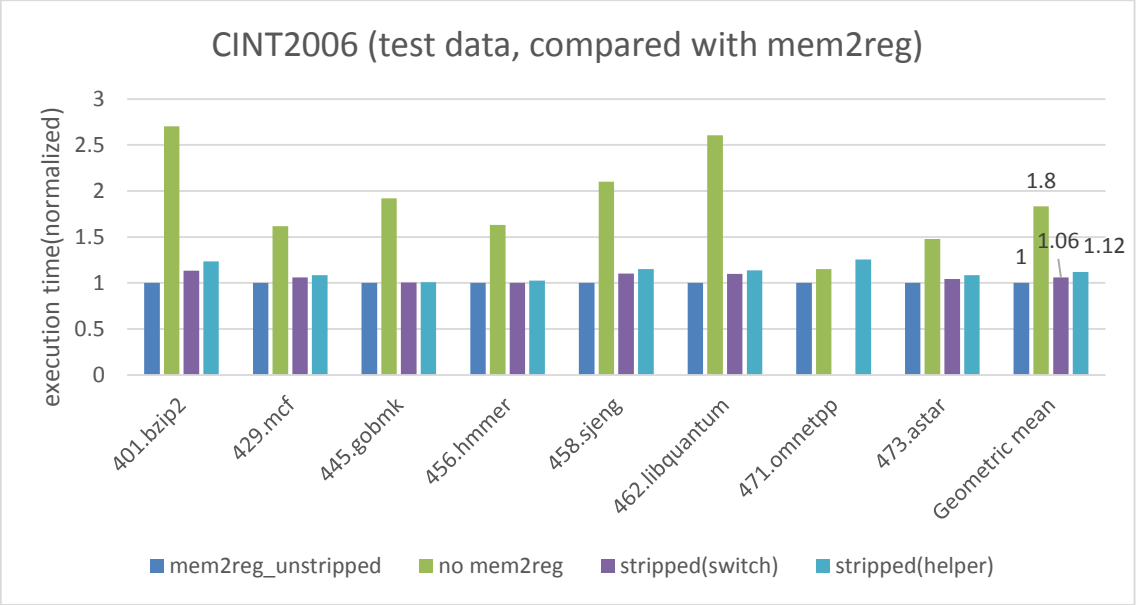


Figure 33. Result of CINT2006 with test data, compared with our best result

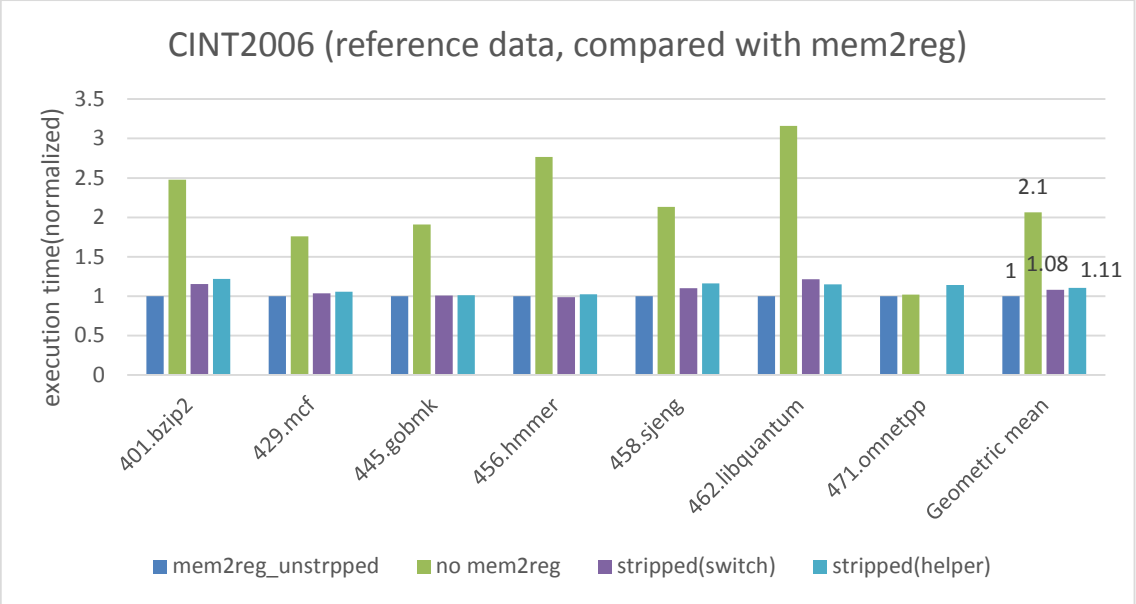


Figure 34. Result of CINT2006 with ref data, compared with our best result

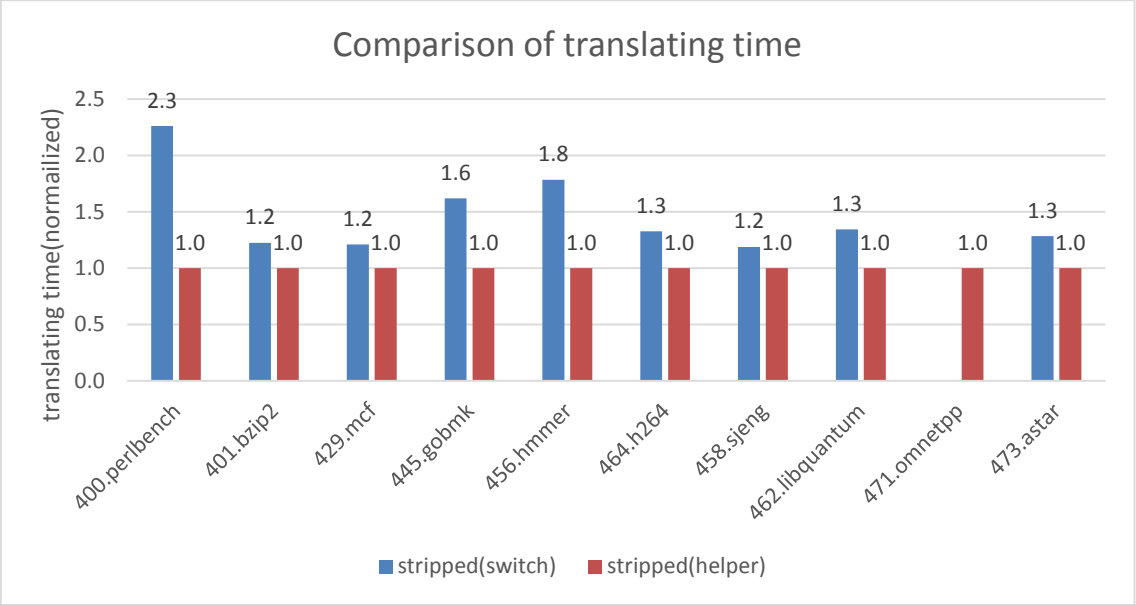


Figure 35. Comparison of translation time when handling stripped function

We list some statistical result of CINT2006 in Table 5, and some benchmarks that are not runnable are also listed in this table. Note that the primary address mapping table of stripped input binary is only about 10% bigger that unstripped one, so the method used for maintaining the secondary one is very important. We give two approaches here, and there must be better method to handle address mapping table, hence, the performance with stripped input binary can be certainly enhanced.

Table 5. Statistical information about CINT2006

	Total	AMT	AMT-stripped	Secondary	Ratio of AMT
400.perlbench	219979	18153	19797	10658	1.09
401.bzip2	21371	1165	1364	732	1.17
403.GCC	589937	54678	58545	25453	1.07
429.mcf	12789	878	1038	502	1.18
445.gobmk	160832	14436	17144	4860	1.19
456.hmmer	66438	7062	7581	2143	1.07
464.h264	35817	2336	2631	1463	1.13
458.sjeng	19739	1914	2120	629	1.11
462.libquantum	119768	6063	6684	3026	1.10
471.omnetpp	142032	27260	29589	5079	1.09
473.astar	25916	2334	2744	1008	1.18

The ratio of the address mapping table and the best case, which is generated by symbol

table, in CINT2006 is much lower than what is EEMBC. Table 6 lists the ratio of the number of return address and the size of address mapping table. Since return address is a constant in both cases, this comparison is reasonable. We can find that the larger the code size is, the higher the ratio of the number of return addresses is. Therefore, we can conclude that the return addresses dominate the size of the address mapping table.

Table 6. Comparison between the ratio of Return address and AMT

	Total	Return	AMT	Ratio	AMT-stripped	Ratio
400.perlbench	219979	16156	18153	89%	19797	82%
401.bzip2	21371	896	1165	77%	1364	66%
403.gcc	589937	50281	54678	92%	58545	86%
429.mcf	12789	653	878	74%	1038	63%
445.gobmk	160832	11660	14436	81%	17144	68%
456.hmmer	66438	6290	7062	89%	7581	83%
464.h264	35817	1965	2336	84%	2631	75%
458.sjeng	19739	1588	1914	83%	2120	75%
462.libquantum	119768	5285	6063	87%	6684	79%
471.omnetpp	142032	23870	27260	88%	29589	81%
473.astar	25916	1873	2334	80%	2744	68%

Partition L-function technique has been described in 3.3.4, and we show some performance results using different parameters, like different partition method, different method for function switching, and different recursive time permitted. The larger the size of L-function, the longer compiling time and shorter execution time. For convenient, we set the size of each slice of L-function being twenty thousand. Furthermore, all of the results are normalized by all best result.

1) Partition method: DFS vs. uniform, in Figure 36.

The results of using DFS to partition functions seem better in most of cases, especially in 471; however, 458 has better performance when using uniform partition method, so there are not definitely correct partition method, we just choose the one that has higher probability to have good performance.

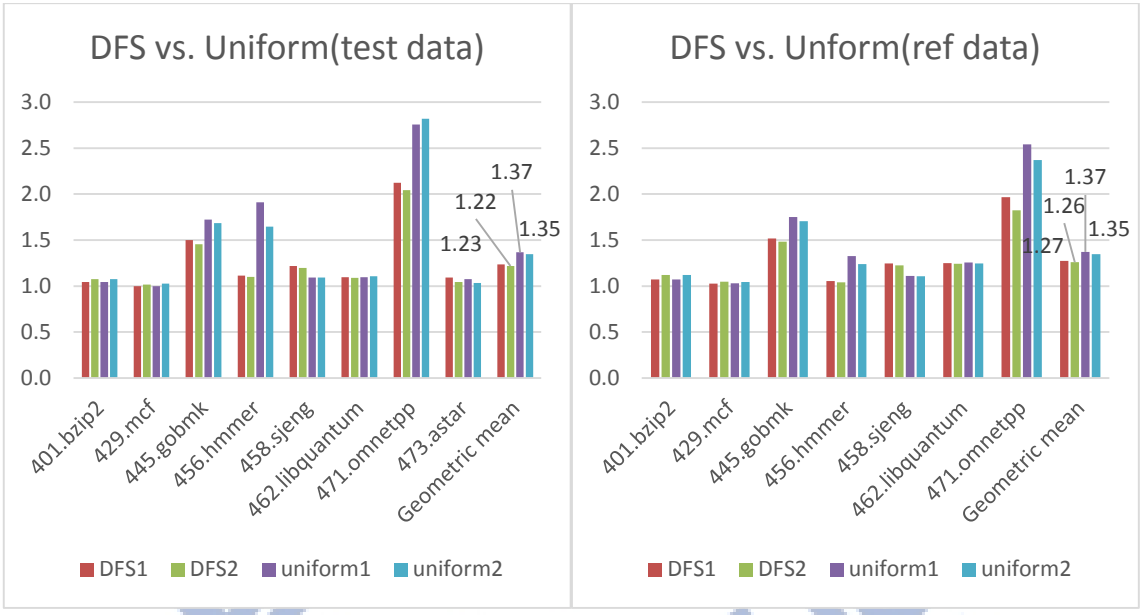


Figure 36. DFS vs. Uniform

2) Method for function switching: helper function vs. LLVM switch instructions, in Figure 37.

In this case, we can't say any one of choice is better. The average shows that using switch instruction is better, but the difference is very small. Therefore, choosing the one that spends less time to translate is better.

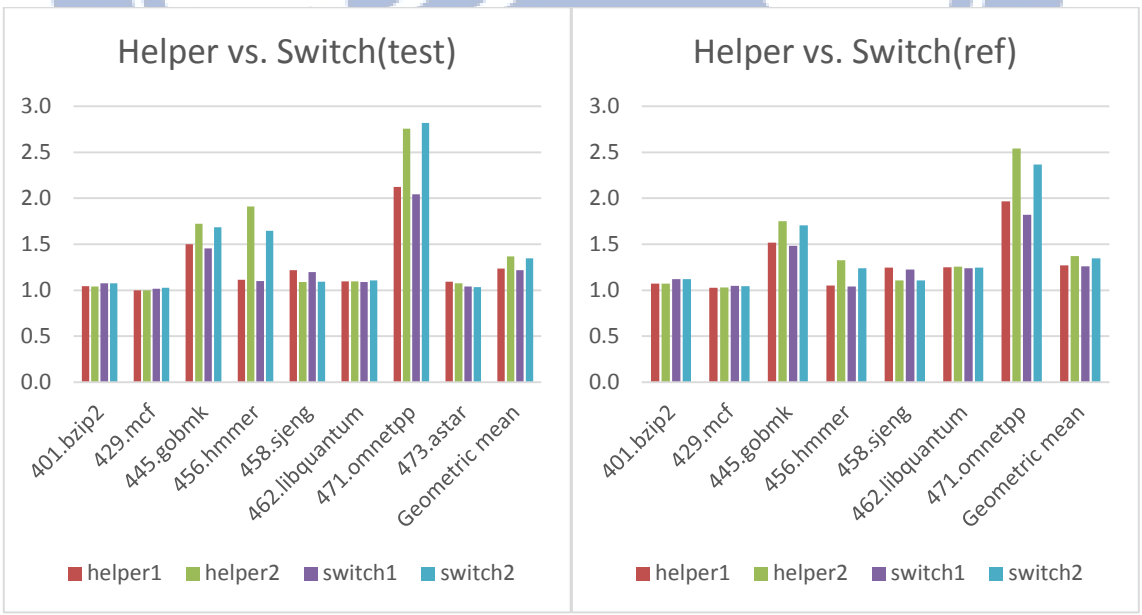


Figure 37. Helper function vs. LLVM switch instruction

3) Recursive times: 0 vs. 2048, in Figure 38.

In this four cases, the results look the same, so whether returning back to the

main L-function every time when function switching is up to the user. The only thing most be noticed is that the maximum recursive times can't be too large; otherwise, the stack may overflow.

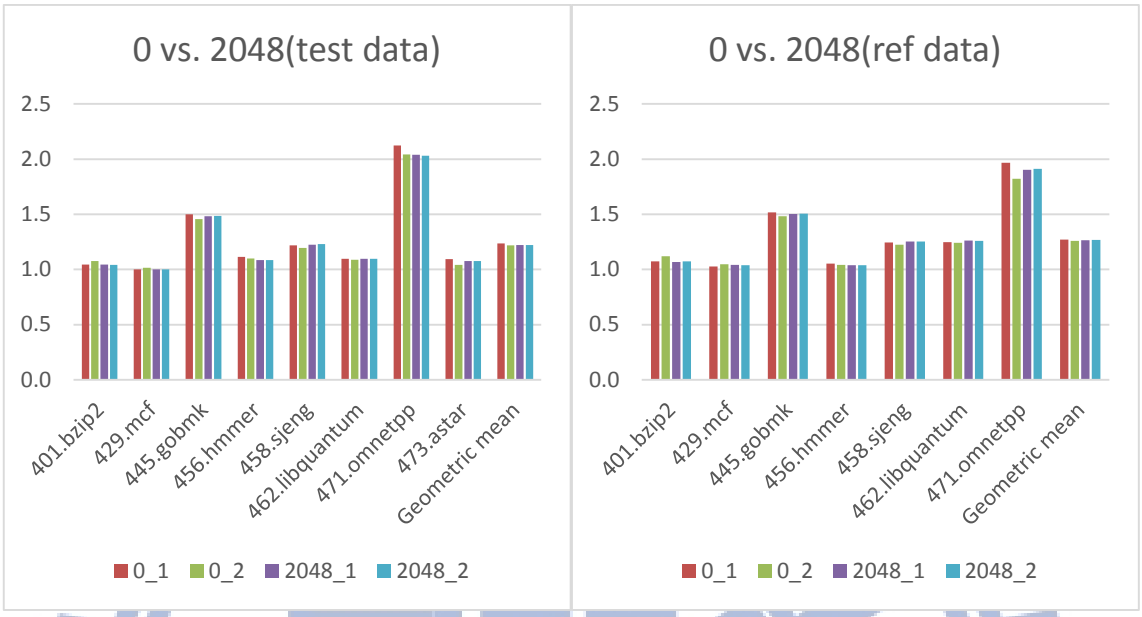


Figure 38. Recursion time comparison: 0 vs. 2048

Actually, four cases in 1) and 2) are the same, except the permutation difference. As a summary, using DFS for partitioning the function and LLVM switch instruction for function switching has more probability to get better performance if the main L-function has to be partitioned in order to save translation time. In fact, the instruction size of 401, 429 and 458 are smaller than twenty thousand, so their data are just for reference.

4.2.2. Translation Time

The translation time indicates the whole time spent in our system. It includes not only the time spent in our translator, but also the time spent in LLVM optimizer and LLVM static compiler. Since code size of EEMBC benchmarks are small, translation time is not that important; otherwise, DBT can be used since the execution time is short. In this section, we show how function-partition lower the translation time of the input binaries.

The only thing we want to demonstrate is that partition the main L-function into several slices of L-function definitely lower the translation time, or more exactly, the compiling time used by LLVM infrastructure, so all of the input binaries are unstripped, and we can use their symbol table to generate the address mapping table.

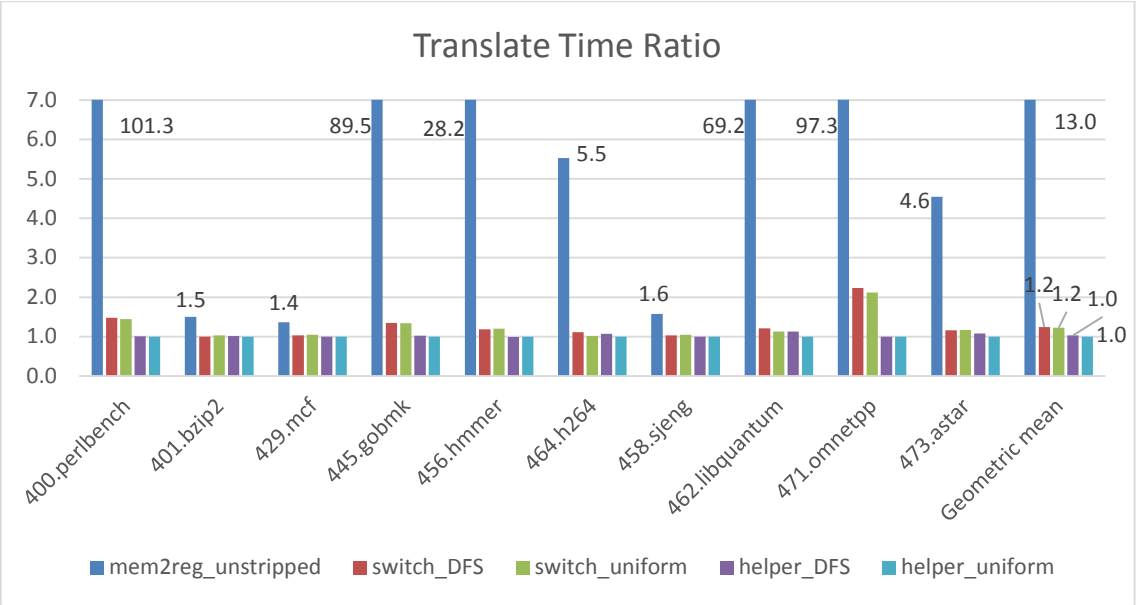


Figure 39. Translation time Ratio

We list four kinds of cases to compare the translation time in Figure 39, and normalize with the lowest one, which uses uniform partition method and calls helper function to handle function switching. This cases takes the shortest time is not strange, since uniform partition method is much easier compared with DFS, and calling helper function prevents too much time spend on optimizing by LLVM infrastructure. The translation time of all benchmark decrease tremendously, except 401, 429, and 458, whose instruction size is too small to be partitioned. Actually the translation time may easily be influenced by other program executing in the same system, so the value in Figure 39 is just for reference. We just claim that the enhancement saves a lot of translation time.

4.3. Code Size

The code size of the target binary must be much more than the code size of the binary for native system since address mapping table has to be embedded in the target binary and it can't be stripped. Code size of EEMBC benchmark is too small, about 300KB for target binary and 50KB for native binary, so the comparison between them is not we concern about.

We list the code size of CINT2006 in Figure 41. Label "stripped switch" indicates that using switch table to access secondary address mapping table, and the input binary is stripped, while "stripped helper" indicates using helper function with indirect branch instruction to access. From the above figure of Figure 41, which based on native binary, we

can see that the flag –mem2reg not only enhance performance but it also decrease the code size, and we can regard the code size without the option –mem2reg as the upper bound since it is the highest one if only one L-function being used. Two cases with multi-L-functions must be larger than others, since they should maintain two mapping table, one for address, and one for function switching.

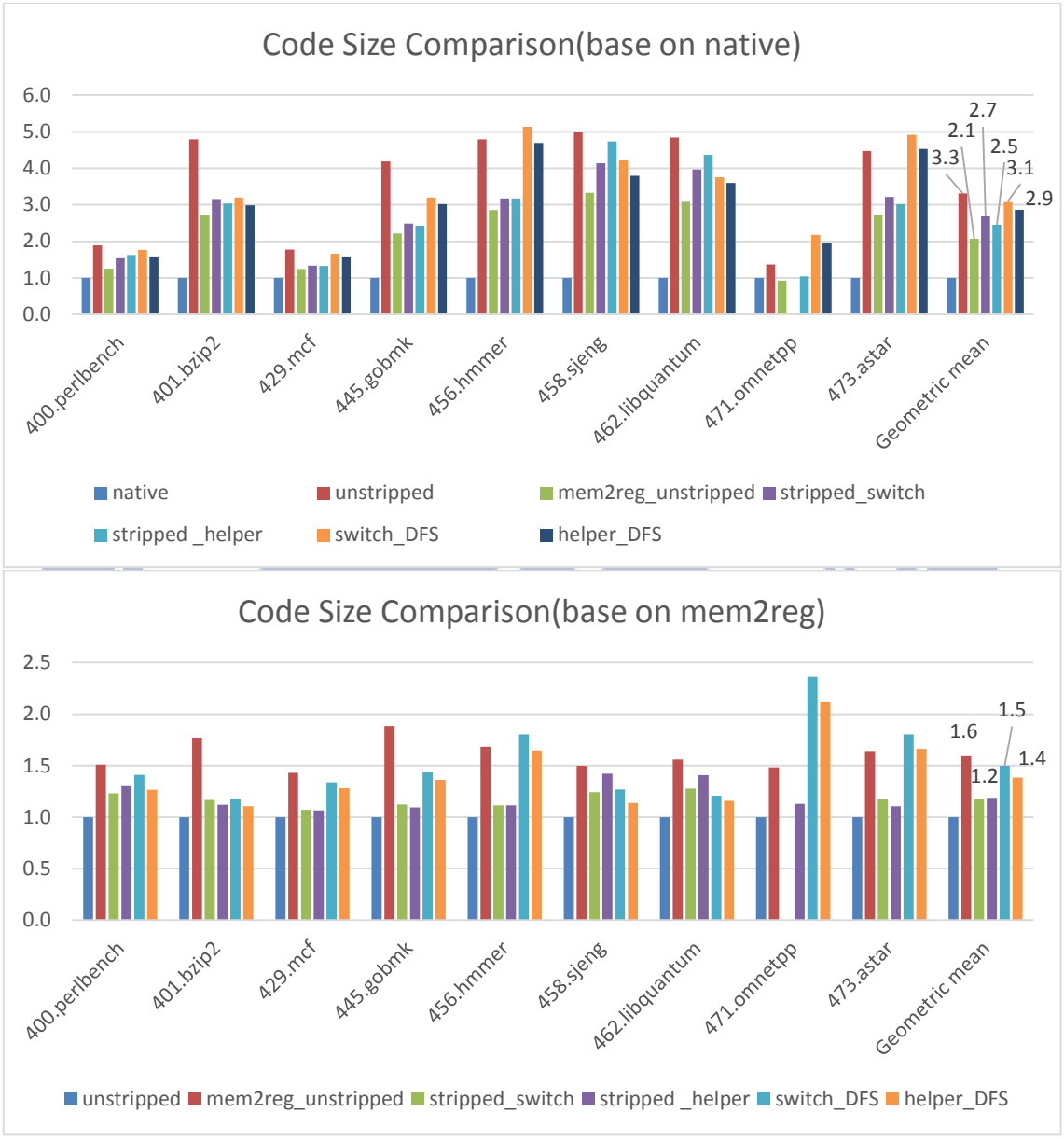


Figure 40. Code size comparison in CINT2006

V. Conclusion and Future Work

In this thesis, we propose a mechanism to analyze the Thumb-2 binaries and locate all types of embedded data in the executables. This work effectively addresses the code-discovery problem for translating GCC generated Thumb-2 executables statically. We also discuss how to expand this work to more general cases as possible future works in section 3.4.

Our implementation is based on the static translator part of Mc2llvm, which is a retargetable hybrid binary translator. We locate PC-relative data by identifying LDR-prefixed instructions and switch tables by using a finite state machine that matches the code patterns generated for switch tables by GCC. Furthermore, we reduce the size of the address mapping table by narrowing down possible addresses that can be function entries. The reduced address mapping table in turn yields much better performance for the translated code. In addition, we have also introduced a framework for partition the translated LLVM functions into smaller slices in order to significantly reduce the compile time. Since LLVM IR file cannot be executed directly, our system finally generates x86 executable for performance comparison.

According to our experiments, the code-discovery problem for GCC-generated Thumb-2 binary has been effectively addressed. With our static Thumb-2 binary translator and using SPEC2006 CINT benchmark (translated by GCC to Thumb-2 code) running with the reference input data, the execution time is about 5.6 times faster than executing with QEMU (a popular system virtual machine via dynamic binary translation). The execution time is about 2.1 times slower, with 2.5 times code expansion, when compared with the x86 native binaries of SPEC2006 CINT translated by GCC. When compared with the results of unstripped Thumb-2 executables, whose function entries can be easily identified by debug symbols, the slow-down of using a little larger address mapping table is only 11% and the overall code expansion is about 20%. Note that, many modern released application binaries are stripped rather than unstripped. Furthermore, with our function partitioning approach, the execution time will be increased by 30% while the translation time could be 13X better if the source is an unstripped executable.

To lower the difference of the execution time between the executable generated by our system and the native executable, more optimization pass should be added when optimizing, including target-independent level (LLVM optimizer) and target-dependent level (LLVM static compiler), or our system should have ability to do some aggressive optimizations when translating.

Our current work is for translating Thumb-2 only executables. One future work could be building a retargetable translator for ARM/Thumb-2 mixed ISA. By combining the mechanisms used in [17] and our work presented in this thesis, a static binary translator for ARM/Thumb-2 mixed ISA would be more practical in general.

Reference

[1] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, "Binary translation," *Commun, ACM,* vol. 36, pp. 69-81, Feb 1993.

[2] J. Y. Chen, W. Yang, C. Su, and W. C. Hsu, "A Static Binary Translator for Efficient Migration of ARM based Applications," in *Proceedings of the 6th Workshop on Optimizations for DSP and Embedded Systems*, 2008.

[3] A. Chernoff, M.Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. Yadavalli, and J. Yates, "A profile-directed binary translator," *IEEE Micro,* vol. 18(2), pp. 56-64, 1998.

[4] B. Y. Shen, J. Y. Chen, W. C. Hsu, and W. Yang, "LLBT: an llvm-based static binary translator," in *In Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems, CASES*, New York, NY, USA, 2012.

[5] B. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling," *ACM,* vol. 22, 1994.

[6] V. Bala, E.Duesterwald, and S. Banerjia, "Dynamo: a transparent dynamic optimization system," *SIGPLAN,* vol. 35(5), pp. 1-12, May 2000.

[7] D.Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *In Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, CGO*, Washington, DC, USA, 2003.

[8] J. Smith and R. Nair, Virtual Machine: Versatile Platforms for Systems and Processes, Morgan Kaufmann, 2005.

[9] C. Cifuentes and V. M. Malhotra, "Binary Translation: Static, Dynamic, Retargetable?," in *Proceedings of the 1996 International Conference on Software Maintenance*, Washinton, DC, USA, 1996.

[10] Bor-Yeh Shen, Jyun-Yan You, Wuu Yang, and Wei-Chung Hsu, "An LLVM-based hybrid binary translation system," in *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, Karlsruhe, Germany, 2012.

[11] "ARM-The Architecture for the Digital World," [Online]. Available: http://www.arm.com/.

[12] "ARM Architecture Reference Manual ARMv7-A and ARMv7-R Edition," [Online]. Available: https://silver.arm.com/download/download.tm?pv=1203633.

[13] "GCC," [Online]. Available: http://gcc.gnu.org/.

[14] 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.

[15] F. Bellard, "QEMU: a fast and portable dynamic translator," in *In proceedings of the annual conference on USENIX Annual Technical Conference, ATEC*, Berkeley, CA, USA, 2005.

[16] Horspool, R. N. and N. Marovac, "An Approach to the Problem of Detranslation of Computer Programs," *Computer Jounal (August),* pp. 223-229, 1980.

[17] Jiunn-Yeu Chen, Bor-Yeh Shen, Quan-Huei Ou, Wuu Yang and Wei-Chung Hsu, "Effective code discovery for ARM/Thumb-1 mixed ISA binaries in a static binary translator," in *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'13)*, Montreal, Canada, 2013.

[18] C. Lattner, "Intro to the LLVM MC Project," [Online]. Available: http://blog.llvm.org/2010/04/intro-to-llvm-mc-project.html.

[19] "EEMBC," [Online]. Available: http://www.eembc.org.

[20] "µClibc," [Online]. Available: http://www.uclibc.org.

[21] R. Spencer and G. Henriksen, "LLVM's Analysis and Transform Passes," [Online]. Available: http://llvm.org/docs/Passes.html.

[22] "SPEC CPU2006," [Online]. Available: http://www.spec.org/cpu2006/.