國 立 交 通 大 學

資訊科學與工程研究所

碩 士 論 文

藉由暫存器配置與指派演算法減少程式碼大小於混合寬度指令集架構處理器

Reducing Code Size by Graph Coloring Register Allocation and Assignment Algorithm for Mixed-Width ISA Processors

研 究 生：王志先

指導教授：單智君 博士

中 華 民 國 九 十 八 年 七 月

藉由暫存器配置與指派演算法減少程式碼大小於混合寬度指令集架構處理器

Reducing Code Size by Graph Coloring Register Allocation and Assignment Algorithm for Mixed-Width ISA Processors

研 究 生：王志先　　Student：Jyh-Shian Wang

指導教授：單智君　　Advisor：Dr. Jyh-Jiun Shann

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 and Engineering

July 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年七月

藉由暫存器配置與指派演算法減少程式碼大小

於混合寬度指令集架構處理器

學生：王志先 指導教授：單智君 博士

國立交通大學資訊科學與工程研究所碩士班

摘要

由於現今的嵌入式系統需要越來越多的程式功能，但又不希望增加記憶體大小，因此減少程式碼大小即成為一個關鍵性的問題。其中一種解決辦法是使用"混合寬度指令集架構"，這類架構通常包含一個正常寬度指令集(通常是32位元)以及一個短指令集(通常是16位元)，且短指令集僅有部分Opcodes以及僅能存取部分的暫存器。在以往傳統的混合指令集架構中，一段連續程式碼僅能被編碼在相同的格式(寬度)，無法使用多種格式穿插其中，但越來越多的混合指令集架構使用了指令編碼來告知處理器該指令的寬度，如此便可在程式之中任意穿插長短指令，不再是一個一個分開的區塊。對於這樣的架構，有多少指令能夠被編碼成較短的格式高度依賴於如何配置這些短指令格式能存取到有限的暫存器。在這篇論文中，我們提出了兩個基於著色演算法的暫存器配置與分派演算法，它們使用一個估計的方法去找出適合被指派到短指令格式能存取到之暫存器的程式變數，而適合的變數意味著如果指派它們到這些暫存器可以有效增加可以被編碼成短指令的指令數量。透過模擬結果顯示，使用此論文所提出的演算法可以減少大約31.90%的程式碼大小。

Reducing Code Size by Graph Coloring Register Allocation and Assignment Algorithm for Mixed-Width ISA Processors

Student: Jyh-Shian Wang Advisor：Dr. Jyh-Jiun Shann

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

Reducing program size is a critical issue in many embedded systems which require more program functionalities without increasing the memory size. One of the approaches is the “mixed-width instruction set architecture (ISA)” which usually has an instruction set in general formats (usually 32-bit long) as normal instruction set, and an instruction set in shorter format (usually 16-bit long) with limited opcodes and set of registers. Traditionally, a code segment can be encoded in only one format, no multiple formats interleaved. However, more and more processors use instruction encoding to indicate the length of each individual instruction, and take mixed-width ISA into instruction-level granularity. For this kind of ISAs, the number of instructions can be encoded in shorter format is highly dependent on the limited set of registers that can be accessed by shorter format instructions. In this paper, we present a register allocation and assignment algorithm based on graph coloring, which uses a heuristic model to find out which virtual variables in program should be assigned into the set of registers accessible by shorter instructions. The simulation results show that the code size reduction is achieved 31.90% by the proposed algorithm.

致謝或序言

首先感謝我的指導老師 單智君教授，在這兩年當中不論是正式報告，亦或是平日小組討論，老師對於學生的諄諄教誨，細心指導與勉勵，使我學習到如何面對問題，以及如何克服問題，並培養獨立研究的能力。有幸跟著老師做研究，觀察老師對於一件事情的執著與細膩，耳濡目染，並從中學習，最後完成了研究與碩士學位。同時，也感謝口試委員，楊武教授與雍忠教授，由於教授們的指導與建議，才使得此篇論文更加完整與充實。另外，也謝謝實驗室的另一位老師，鍾崇斌教授，在一次次的報告之中給予學生指導與建議。

裕生學長、奕緯學長，感謝兩位學長帶領我進入 JVM 與 Compiler 領域，不僅僅只是給予我研究上的建議與討論，同時也常是鼓勵我前進的動力，才使得我學習到相關知識並完成此研究。同時，實驗室的學長姐、同儕以及學弟妹，在這一起渡過的時光中，你們不僅僅是我記憶中的好夥伴，更是人生道路上的貴人們，謝謝。

最後，對於我的家人以及總是在我低潮時給予鼓勵並且陪伴著我的親友們，志先也在此獻上最誠摯的謝意。

王志先　2009.7.26

Table of Contents

摘要 i

Abstract ii

致謝或序言 iii

Table of Contents iv

List of Figures vi

List of Tables viii

Chapter 1 Introduction 1

1.1 Research Motivation 4

1.2 Research Objective 4

1.3 Organization of this Thesis 6

Chapter 2 Background 7

2.1 Mixed-width ISA with Mode-switch by Instruction Encoding 7

2.1.1 S-Format Limitations 7

2.1.2 Encoding Formats of S-Format Instruction 9

2.2 Graph Coloring Register Allocation 10

2.2.1 Interference Graph 11

2.2.2 Bottom-up Graph Coloring 11

2.2.3 Priority-based Graph Coloring 12

2.3 Summary of Backgrounds 14

Chapter 3 Design of The Register Allocation and Assignment Algorithms 15

3.1 Compiler Back-end and Definitions for Mixed-width ISA 16

3.1.1 Instruction Types 18

3.1.2 Register Classes 19

3.2 Design I : Based on Bottom-up Graph Coloring 20

3.2.1 Allocation Pass 21

A. Reg_S-Simplify and Reg_L-Simplify Stages 21

B. Spill Stage 24

3.2.2 Assignment Pass 25

3.2.3 Discussion ...28
A. Different Assignment Orders in Assignment ...28
B. Assignment without Reg_S-Simplify ...31
C. Extension of the Algorithm to More Hierarchy Register Sets for Different S-Formats ...32
3.3 Design II : Based on Priority-based Graph Coloring ...34
3.3.1 Separate Stage ...37
3.3.2 LR_U Allocation and Assignment ...37
3.3.3 LR_L Allocation and Assignment ...38
3.3.4 LR_S Allocation and Assignment ...39
3.3.5 Discussion ...39
A. Alternative Design ...40
B. Extension of the Algorithm to More Hierarchy Register Sets for Different S-Formats ...41
Chapter 4 Experiment ...43
4.1 Environment ...43
4.2 Benchmark Evaluation Results ...44
4.2.1 Parameter Determination ...44
4.2.2 Comparisons of Design Alternatives ...46
4.2.3 Code Size Reduction ...47
4.2.4 Spill Codes ...50
4.2.5 S-Format Limitations Analysis ...51
4.3 Summary for Simulation Results ...53
Chapter 5 Conclusions and Future Works ...54
5.1 Conclusions ...54
5.2 Future Works ...55
References ...57

List of Figures

Figure 1-1 – L/S-Format instructions in program code. (a) Mode-switch by mode-switch instruction. (b) Mode-switch by instruction encoding. 3
Figure 1-2 – Mode-switch by instruction encoding. 3
Figure 2-1 - S-Format limitation distributions. 8
Figure 2-2 - Different encoding formats of S-Format instructions. (Rd: destination register. Rs, Rt: source destination. Imm: immediate value) 9
Figure 2-3 - Interference graph. 11
Figure 2-4 - Bottom-up graph coloring. 12
Figure 2-5 - Priority-based graph coloring. 13
Figure 3-1 - Compiler back-end for mixed-width ISA. 17
Figure 3-2 - Different S-Format instructions equivalent to *lw*. 17
Figure 3-3 – Register classes 19
Figure 3-4 - Flowchart of the proposed algorithm based on Bottom-up graph coloring for mixed-width ISA with mode-switch by instruction encoding. 20
Figure 3-5 – Diamond graph. 22
Figure 3-6 - The process sketch map of our proposed algorithm based on Bottom-up graph coloring. 22
Figure 3-7 - Pseudo code of allocation pass. 24
Figure 3-8 - Pseudo code of assignment pass. 28
Figure 3-9 - Different Assignment Order in Bottom-up Graph Coloring for Mixed-width ISA with mode switch by instruction encoding : (a) Reg_S-Assignment → Reg_L-Assignment (b) Reg_L-Assignment → Reg_S-Assignment. 30
Figure 3-10 - Assignment result without Reg_S-Simplify 31
Figure 3-11 - Extension of Design I for different S-Formats. 33
Figure 3-12 - Flowchart of the register allocation and assignment algorithm based on Priority-based graph coloring for mixed-width ISA with mode-switch by instruction encoding. 35
Figure 3-13 - Pseudo code of Priority-based graph coloring for mixed-width ISA with mode-switch by instruction encoding 36
Figure 3-14 - Flowchart of the proposed algorithm without LR_L allocation and assignment pass of Design II. 40
Figure 3-15 - Extension of Design II for different S-Formats. 42
Figure 4-1 - The evaluation of different α values. 45
Figure 4-2 - Benchmark evaluation results of the different assignment order in MxBuGCRA. 46
Figure 4-3 - Benchmark Evaluation Results of MxPrGCRA with/without LR_L pass. 47

Figure 4-4 - Benchmark evaluation results of the proposed algorithms.................................49
Figure 4-5 - Spill codes of the proposed and traditional algorithms. 50
Figure 4-6 - S-Format limitation distributions - BuGCRA v.s. MxBuGCRA. 52
Figure 4-7 - S-Format limitation distributions - PrGCRA v.s. MxPrGCRA...........................52

List of Tables

Table 4-1 - Benchmark Evaluation Results of the Proposed Algorithms................................49

Chapter 1 Introduction

In the increasing market of embedded systems, RISC processors have been used widely. A RISC processor usually offers higher computation power and lower hardware cost and, meanwhile, suffers from the less code density than a CISC processor because of its fixed-width instruction set. However, code size is one of the major issues in embedded systems, since the larger code size may increase the memory requirement. As a result, mixed-width RISC instruction set architectures (ISAs) have been proposed to make a good tradeoff between performance and code density (i.e. low code size) [1]. Moreover, the traffic of the memory data bus for fetching instructions and the I-Cache miss rate may also be reduced.

There are several mixed-width ISAs provided commercially, for examples, ARM's ARM/Thumb ISA, MIPS' MIPS/MIPS16 ISA, Andes' AndeStar ISA, etc [2-4]. They typically have one short width instruction format (S-Format) as a frequently used subset of the longer width instruction format (L-Format). For example, MIPS is a 32-bit width instruction set, and its 16-bit width subset is called MIPS16. Mixing the short width instructions into the original program which is composed with 32-bit instructions may improve the code density. However there are two main limitations exists due to the S-Format instructions have fewer bits for register indexing and immediate value storing in mixed-width ISAs.

1. **Fewer bits to index registers:**

   One of the limitations is that the short width instructions have fewer bits to index registers. For example, 3-bit register field in S-Format can access eight physical registers only. If all of the operands of an instruction are assigned to the registers that can be accessed by S-Format instructions, then this instruction is able

to be encoded as an S-Format instruction to reduce code size. Otherwise, if one of its operands is out of the register indexing range of S-Format instructions, then this instruction must be encoded as an L-Format instruction definitely. Accordingly, if the compiler does not take into account these restrictions while assigning registers, the translation rate of S-Format instructions may be quite low. Therefore, the assignment of registers becomes very important for mixed-width ISAs.

2. **Fewer bits to hold immediate values:**

   The other limitation is the short width instructions have fewer bits to store immediate values. If the immediate value is oversized for an instruction's S-Format then the instruction can only be encoded as L-Format instruction. Although large immediate values may impact the translation rate of S-Format instructions, it varies on how complier manages constants. If a compiler uses a constant pool to hold these large immediate values, the impact of immediate values can nearly be neglected.

In addition, the different mechanisms of mode switching between L-Format and S-Format instructions make problems distinct in mixed-width ISAs. There are two types of mechanisms for switching between L-Format and S-Format instructions [5]. Some architectures use a mode switching instruction to change modes between code segments with different encoding formats, for example, ARM/Thumb. It means that all instructions in the same code segment must be encoded in the same format as shown in Figure 1-1 (a). On the other hand, there are some architectures change modes by instruction encoding so that L-Format and S-Format instructions may be interleaved freely in routines as shown in Figure 1-1 (b), i.e., L-Format and S-Format instructions may be mixed up at the instruction level granularity. For example, AndeStar ISA uses a bit (usually the MSB) in instruction field to

indicate whether the instruction is L-Format or S-Format as shown in Figure 1-2. For the former, existing compilers either rely on user guidance or perform an analysis to determine which code segments should use S-Format [6], then a mode switch instruction will be inserted between the code segments, and finally the compiler compiles code segments with different instruction width by different policies. For the latter, because no mode-switch instruction is needed, the compiler should eliminate the limitations of each individual instruction of its S-Format as far as possible to increase the number of instructions encoded in S-Format. However, the existing techniques for this kind of ISAs are still rudimentary.

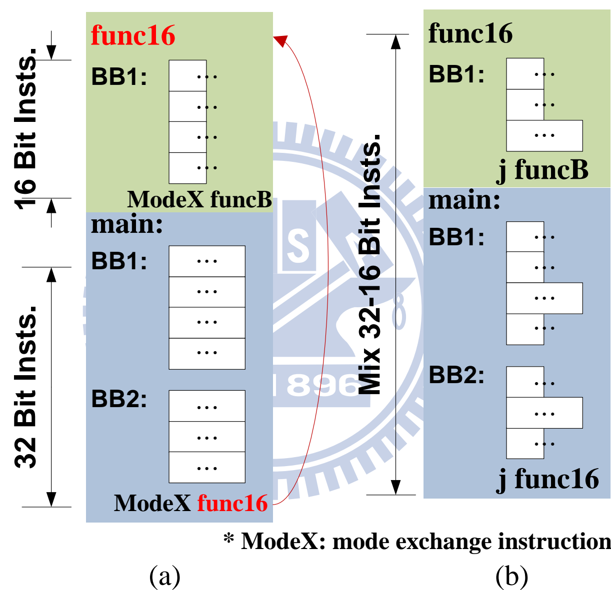


Figure 1-1 – L/S-Format instructions in program code. (a) Mode-switch by mode-switch instruction. (b) Mode-switch by instruction encoding.

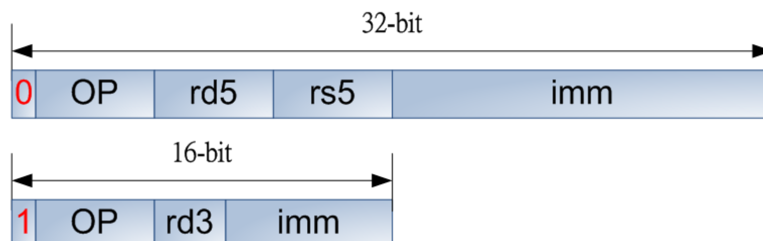


Figure 1-2 – Mode-switch by instruction encoding.

1.1 Research Motivation

So far we have introduced the mixed-width ISAs, which can increase code density if the registers are used carefully especially for those with mode-switch by instruction encoding. Also, we know that the code size problem is one of the major issues in embedded systems. The larger code size needs the larger memory, and thus may consume more power. Unfortunately, the enlarging program size due to the requirement of more program functionalities in modern embedded applications is happening. For these reasons, using mixed-width ISAs is a feasible approach for code size reduction.

In order to reduce program code size for a mixed-width ISA with mode-switch by instruction encoding, registers should be allocated and assigned properly to eliminate each instruction's limitation of translation to S-Format instructions, and, as in results, the number of instructions that can be encoded as S-Format may be increased. However, the existing techniques of compilers for mixed-width ISA with mode-switch by instruction encoding are rudimentary. Therefore, a proper register allocation and assignment algorithms should be designed for this kind of ISAs.

1.2 Research Objective

In this thesis, we proposed an algorithm for mixed-width ISA with mode-switch by instruction encoding to increase the number of instructions encoded as S-Format by allocating and assigning registers properly. The original goal of register allocator is to allocate virtual variables to registers or memory locations and optimize for generating fewest memory referenced instructions (spill codes). However, for the mixed-width ISAs, it should consider

the mapping of physical registers and virtual variables, and, meanwhile, the number of spill codes should be minimized due to the performance issue. To achieve the features mentioned, there are two main goals to accomplish:

1. **Reducing code size:**

   To reduce code size by mixed-width ISA with mode-switch by instruction encoding, the usage of S-Format instructions is the key point. In other words, if we can encode more instructions as S-Format instructions, the code size will be reduced more. To achieve that, we propose a heuristic model in register assignment procedure. The proposed algorithm not only allocates virtual variables to registers or memory locations but also assigns registers by choosing virtual registers with the highest code size benefit to assign physical registers which are accessible by S-Format instructions.

2. **Minimizing performance degradation:**

   Although our primary objective is to reduce code size, the number of spill codes generated by register allocator is critical, too. More spill codes lead to more performance degradation, and it also increases code size. To minimizing the number of spill codes, the proposed algorithm chooses variables with the lowest memory reference cost to spill while the required registers are more than the physical registers available.

1.3 Organization of this Thesis

The rest of this paper is organized as follows: Section 2 discusses more details of a typical mixed-width ISA and other related researches and algorithms; Section 3 gives the instruction formats and register classes defined in our algorithm and the detailed description of the proposed algorithm; Section 4 presents the experimental results and discussion follows. Finally, Section 5 dedicates to the conclusions we draw and the future work planed.

Chapter 2 Background

In the first part of this chapter, we will analyze the distribution of S-Format limitations in programs generated by a traditional register allocation algorithm to observe the opportunities for research and explain more details about mixed-width ISA with mode-switch by instruction encoding. In the second part, two traditional register allocation algorithms will be introduced. In the last part, a brief summary about the problems that the existing algorithms suffer from in mixed-width ISA with mode-switch by instruction encoding and the solutions we proposed for it will be described.

2.1 Mixed-width ISA with Mode-switch by Instruction Encoding

In this section, the S-Format limitations will be introduced and analyzed. Then, the different encoding format of S-Format instructions will be explained in details.

2.1.1 S-Format Limitations

Using mixed-width ISA can reduce code size significantly in intuition: if all instructions have operational equivalent S-Format instructions and all of them can be encoded in S-Format, then the code size may be reduced by 50%. However, as mentioned above, there are many restrictions including register index, immediate values, and, even that not all the instructions have corresponding S-Format instructions.

Figure 2-1 is the analysis result about the distribution of the limitations of S-Format translation in benchmark programs compiled by using traditional graph coloring register allocation. The top blocks, green colored, are the percentage of instructions which have

no operation equivalent S-Format instructions or the immediate value is oversized. The middle blocks, red colored, are the percentage of instructions each of which has one or more operation equivalent S-Format instructions but at least one of its operand registers is out of range to index of S-Format instruction. And the bottom blocks, blue colored, are those instructions whose register number and immediate value are in the range of S-Format instructions.

From the distribution, we found that there are only about 11% instructions which have no operation equivalent S-Format instructions or the immediate value is oversized for its S-Format. However, over 50% of the left 89% instructions are restricted by their register number for translating to S-Format instructions. If the register allocation uses registers that can be accessed by S-Format instructions carefully in a heuristic way, it is possible to make more instructions be encoded into S-Format.

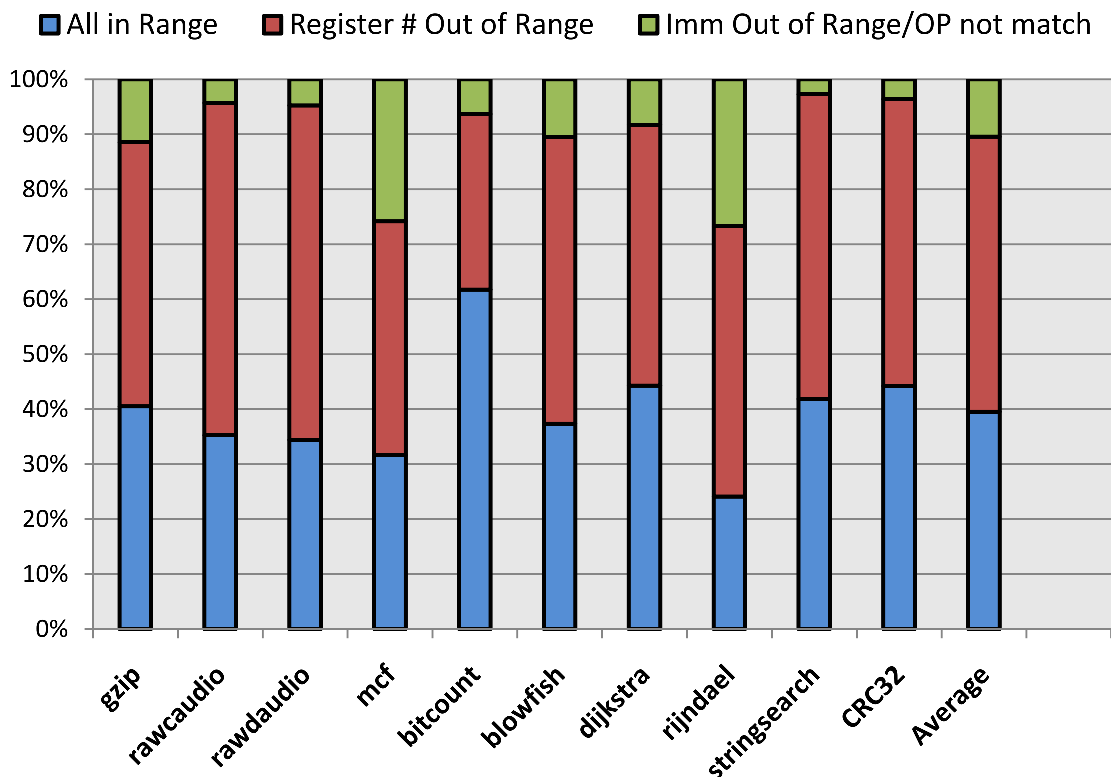


Figure 2-1 - S-Format limitation distributions.

2.1.2 *Encoding Formats of S-Format Instruction*

Comparing with L-Format instructions, S-Format instructions have fewer bits to index registers, and thus, the encoding formats of S-Format instructions have to be designed carefully in mixed-width ISAs. Typically, there are four S-Format encoding forms in mixed-width ISAs in present as in Figure 2-2:

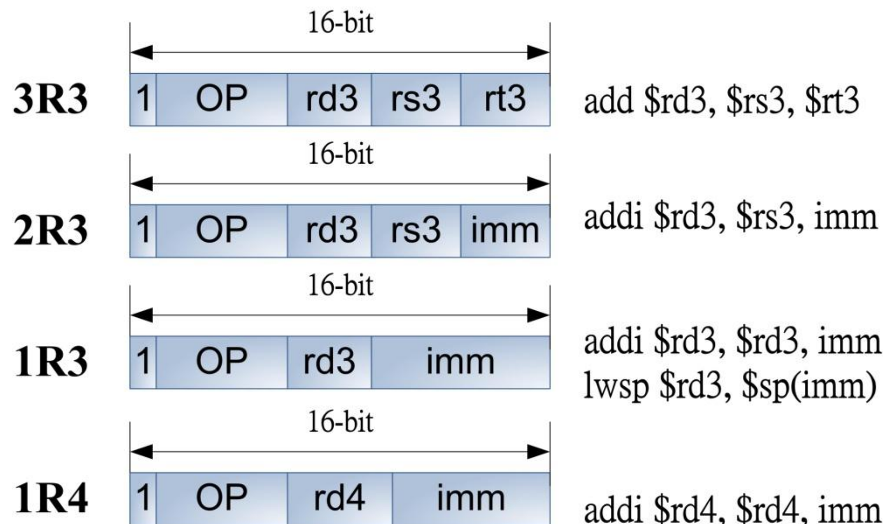


Figure 2-2 - Different encoding formats of S-Format instructions. (Rd: destination register. Rs, Rt: source destination. Imm: immediate value)

1. **3R3 Form**

   There are three registers as its operands, and three bits for each register indexing in 3R3 form.

2. **2R3 Form**

   There are two registers and one immediate value as its operands, and three bits for each register indexing in 2R3 form.

3. **1R3 Form**

   There are one register and one immediate value as its operands, and three bits for register indexing in 1R3 form. The register, rd3, in this form is used as the

source register and the destination register (eg. addi in Figure 2-2) or destination register when there is an implied register (eg. lwsp in Figure 2-2).

4. **1R4 Form**

In a few mixed-width ISAs, some S-Format instructions may use four bits for register indexing. And the register, rd4, in this form is used as the source register and the destination register (eg. addi in Figure 2-2).

So far we have introduced four forms of instruction formats for S-Format instructions in mixed-width ISA with mode-switch by instruction encoding. From the analysis result shown above, the effect of reducing code size by using mixed-width ISAs is highly depend on the number of bits used to index registers. And thus, we concentrate on the number of bits used to index registers, meanwhile, the accessible registers of 3R3, 2R3, and 1R3 forms are limited in the same region. We have surveyed most of the mixed-width ISAs in present, and almost all of them consist of the 3R3, 2R3 and 1R3 forms only. Therefore, in this thesis, our target is the mixed-width ISAs with mode-switch by instruction encoding which comprise 3R3, 2R3, and 1R3 forms. Moreover, we will bring a briefly discussion about how to extend our algorithm to meet the requirement of 1R4 form.

2.2 Graph Coloring Register Allocation

The graph coloring algorithm is the most popular Register Allocator (RA) in general compiler for generating fewest load/store instructions, usually called "spill code". Graph coloring for register allocation has many different versions [7]. The most well-know one is the

Bottom-up Graph Coloring proposed by Gregory J. Chaitin [8][9]. In addition, another frequently used version is the Priority-based Graph Coloring proposed by Fred C. Chow [10]. We will introduce these two algorithms in the following sections.

2.2.1 Interference Graph

Both of these two graph coloring algorithms have to construct the same interference graph for coloring. The interference graph is constructed from a program function as in Figure 2-3. Nodes in an interference graph represent virtual variables (called variables for short) in program function, and the edges connected between nodes is the overlaps between their live ranges, i.e. they are alive in the same time. (Coloring a node in graph is meant giving a register to a variable)

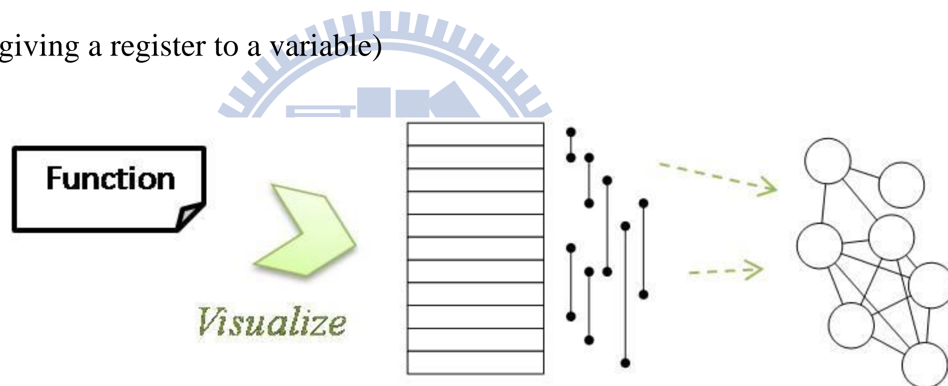


Figure 2-3 - Interference graph.

2.2.2 Bottom-up Graph Coloring

The flowchart of the bottom-up graph coloring algorithm is shown in Figure 2-4. First, it constructs an interference graph in the Build stage. If the graph can be colored with R colors then the variables can be stored in R registers [11]. This algorithm removes a node i with degree (the number of connected edges of a node) less than R, and put i into a stack, iteratively. This stage is called Simplify. Once all nodes are removed from the graph, i.e., the graph is an R-colorable graph, then pops all nodes from the stack and

assigns a color to each of them. Otherwise, the graph is not R-colorable, and thus, chooses one from the remaining nodes and split it into several nodes with shorter live time. This stage is called Spill. Notice that once a variable is spilled, new variables will be produced, and the algorithm must be rewound back to rebuild the interference graph.

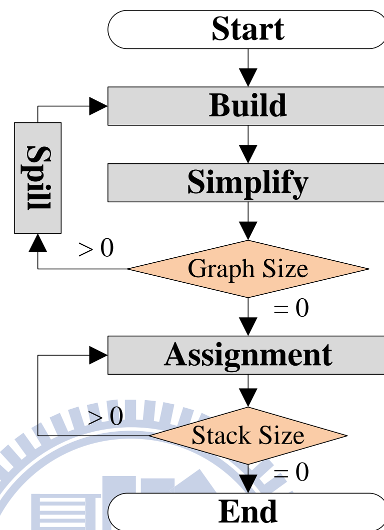


Figure 2-4 - Bottom-up graph coloring.

2.2.3 *Priority-based Graph Coloring*

The flowchart of priority-based graph coloring algorithm is shown in Figure 2-5, it also constructs an interference graph in the Build stage firstly. Then the algorithm separates nodes into two categories: UCLR (Unconstrained Live Range) and CLR (Constrained Live Range). The UCLR contains the nodes with degree less than the available number of physical registers R, and the CLR contains the others. Because the variables in UCLR have degrees (simultaneously living variables) less than the number of available physical registers, all of these variables are guaranteed to have registers. Therefore, the priority-based graph coloring makes effort on the variables in CLR only.

It assumes that all variables in CLR are initially stored in memory, and it uses a priority function to give each variable in CLR a priority. Then pick the highest priority one to assign physical register. The priority function is composed of the cost of memory load and memory store saved if the variable was assigned to a register. In other words, the priority of a variable is similar to the access counts for the variable. After assigning one variable, the algorithm will check each neighbor of the variable to see whether it needs to be spilled. If yes, then spill it. Until all variables in CLR have been assigned registers, priority-based graph coloring then handles the UCLR. As mentioned above all variables in UCLR can be easily assigned registers.

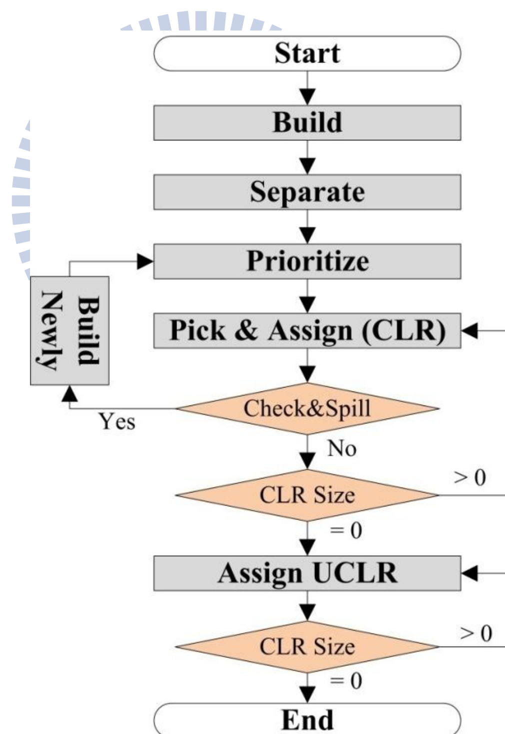
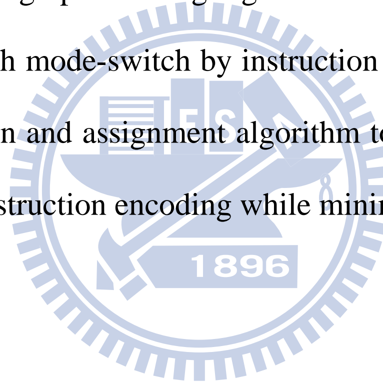


Figure 2-5 - Priority-based graph coloring.

2.3 Summary of Backgrounds

For mixed-width ISA with mode-switch by instruction encoding, we have indicated that the most limitation of translation to S-Format instructions is the using of the accessible registers in the register file in Section 2.1. Therefore, it is critical that which virtual registers should be assigned to which physical register in such architectures. However, both of the two traditional graph coloring register allocation algorithms consider the number of spill code only, and they do not take the mapping of virtual variables and physical registers into account. Therefore, they may result in less translation rate of S-Format instructions in generated code.

Obviously the traditional graph coloring register allocations do not suit for reducing code size in mixed-width ISA with mode-switch by instruction encoding. In this thesis, our goal is to design a register allocation and assignment algorithm to reduce code-size for mixed-width ISA with mode-switch by instruction encoding while minimizing performance degradation.



Chapter 3 Design of The Register Allocation and Assignment Algorithms

In this thesis, we propose a register allocation and assignment algorithm to determine which of the virtual registers (variables) should be in which physical register or memory at each execution point. Here, register allocation decides which variables should be kept in the physical registers, and register assignment chooses physical registers for those variables which are not spilled into memory. In order to minimize the code size, the goal of the proposed algorithms is to assign as many variables which have more benefit to reduce code size (called $\$Var_{CS}$ for short) as possible into the S-Format accessible registers. Therefore, more instructions can be translated to S-Format instructions. Since it is difficult to assign all $\$Var_{CS}$s within an application to the S-Format accessible registers, the way to determine which $\$Var_{CS}$ should be assigned to these registers is crucial. The basic idea of the proposed algorithms is to let the frequently accessed and lower degree $\$Var_{CS}$s have higher opportunity to be assigned to the S-Format accessible registers. In other words, the assignment order of $\$Var_{CS}$ is arranged from the most to the least frequently accessed and from lower to higher degree.

The graph coloring algorithm is the most commonly used register allocation algorithm in compilers so that we use graph coloring algorithm as the base for our algorithms. In the following sections, the compiler backend and definitions used in this thesis are presented first, and then two algorithms which are modified from bottom-up and priority-based graph coloring algorithms for mixed-width ISA with mode-switch by instruction encoding will be explained in detail.

3.1 Compiler Back-end and Definitions for Mixed-width ISA

Typical compiler back-ends consist of instruction selection, instruction scheduling, and register allocation. First, instruction selection maps low-level intermediate representation (IR) to actual machine instructions (called instruction for short). This is usually done by pattern matching. And then, instruction scheduling schedules instruction for hiding some pipeline stall and/or increasing instruction-level parallelism. Finally, the register allocator will allocate virtual registers to physical registers.

However, in a mixed-width ISA, an instruction may be represented in multiple formats (e.g., L-Format and S-Format in this thesis) that are mainly different in their encoding length and the access range of register file. Therefore, instruction selection in a mixed-width ISA not only maps an operation of low-level IR (called operation for short) to instruction(s) but also chooses a proper format for each instruction. To achieve this, instruction selection is separated into two passes in this thesis as shown in Figure 3-1.

In instruction selection pass, an operation is mapped to temporary instruction(s), called as INS. Note that each INS may have multiple instruction formats which are only different in their encoding length rather than functionality. The instruction formatting pass is performed after register allocation and assignment pass. In this pass, each INS is translated to a proper instruction format according to the result of register allocation and assignment. For example, if a memory load INS *lw* has two equivalent S-Format instructions as shown in Figure 3-2. The instruction formatting pass will check whether its base register is $sp. If yes, and the offset is not oversize for S-Format to encode, and, thus, instruction formatting translates *lw* into *lwsp*; if not, the operation will be leaved unchanged. Because the assembly name of

S-Format instruction *lw* is the same as that of the L-Format one, the assembler will check the operands of the instruction to encode it in a proper format.

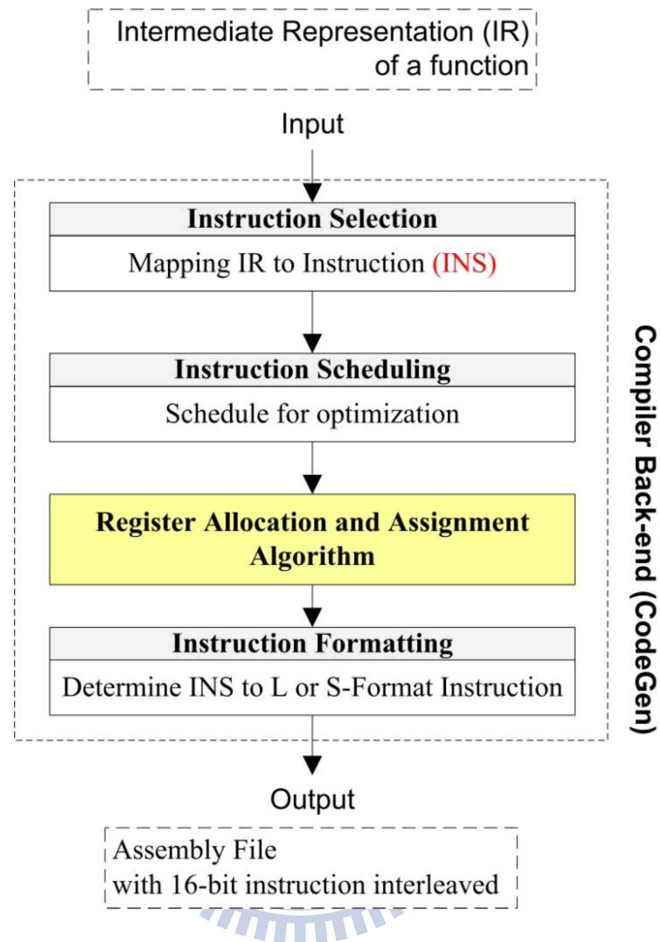


Figure 3-1 - Compiler back-end for mixed-width ISA.

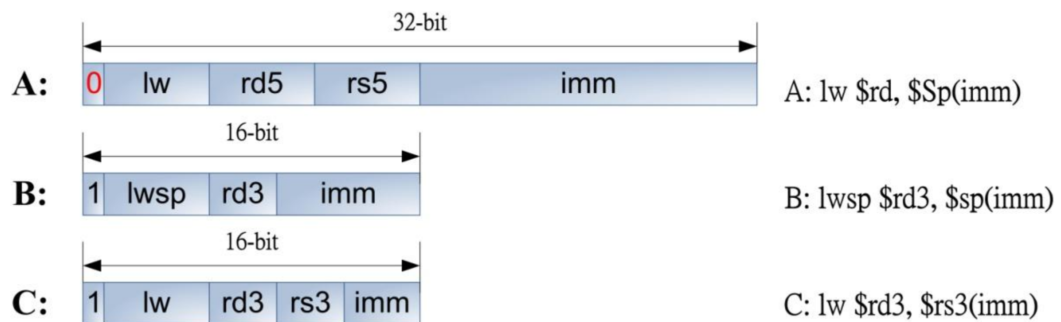


Figure 3-2 - Different S-Format instructions equivalent to *lw*.

3.1.1 Instruction Types

In mixed-width ISA, most INSs have multiple formats that are mainly different in their encoding length. In order to indicate whether an INS has multiple encoding formats or not, in this paper, INSs are classified into three categories as follows:

(1) **L-INS (L-Format INS):** L-INS is an INS which has no equivalent S-Format instruction or has larger (oversized) immediate value such that it cannot be encoded in S-Format.

(2) **S-INS (S-Format INS):** S-INS is an INS which can be encoded in S-Format definitely. For examples, JR and NOP. JR has only one operand and can access anyone of the physical registers. NOP does not index any registers and, thus, can be encoded in S-Format obviously.

(3) **U-INS (Uncertain-Format INS):** U-INS is an INS which has one or more equivalent S-Format instructions, and can be encoded as an S-Format instruction if its physical registers may be limited in a small range. For specification, the virtual variables defined and/or used by a U-INS are marked as $\$Var_U$. And the $\$Var_U$ is exactly that variable with benefit to reduce code size as we mentioned $\$Var_{CS}$. Note that U-INS is exactly what we can make efforts to increase the translation rate of S-Format instructions.

3.1.2 Register Classes

In this thesis, the physical registers of a mixed-width ISA are divided into two classes. Registers that can be accessed by S-Format instructions of U-INSs are denoted as $RegisterSet_S$, and another case is $RegisterSet_A$, which represents all accessible physical registers for L-Format instructions, except some special register such as stack pointer (SP), global pointer (GP), etc. The number of registers of $RegisterSet_S$ and $RegisterSet_A$ are denoted as RSN_S and RSN_A, respectively. Noticeably, $RegisterSet_S$ is a subset of $RegisterSet_A$ rather than independent of each other. Now we take MIPS/MIP16 mixed-width architecture as an example. The register r0 is reserved for zero in MIPS, and MIPS16 uses three bits to index registers. Therefore, $RegisterSet_S$ contains r1~r7, and another class, $RegisterSet_A$, contains r1~r27 as shown in Figure 3-3.

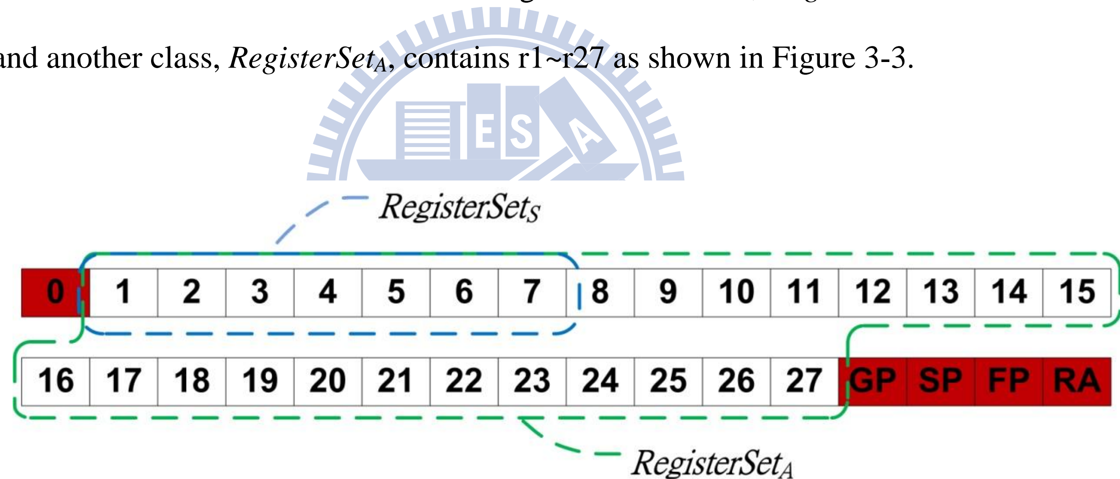


Figure 3-3 – Register classes

Note that we do not handle the special registers, because special registers are usually used in a specific way, e.g., stack pointer ($sp) in *add* or *lw* for stack operation. Moreover, special registers usually have corresponding special S-Format instructions for them as implied operands. For example, *add (SP-relative)* and *lw (SP-relative)* instructions imply SP register as their operand as shown in Figure 3-2.

3.2 Design I : Based on Bottom-up Graph Coloring

In this section, the first design, register allocation and assignment algorithm based on Bottom-up graph coloring for mixed-width ISA with mode-switch by instruction encoding, will be described in detail.

As depicted in Figure 3-4, the proposed algorithm consists of two main passes, namely Allocation, and Assignment. Firstly, the Build stage parses all necessary information, such as live rages, instruction types, etc., and constructs the interference graph, called graph for short. Note that the graph does not contain the variables which must be allocated in special registers and calling convention registers. Based on the interference graph, Allocation pass determines which variable should be allocated in the physical register or be spilled into memory. Finally, Assignment pass selects a physical register and assigns it to a non-spilled variable. The input and output of the proposed algorithms are INSs with virtual registers and assembly code, respectively.

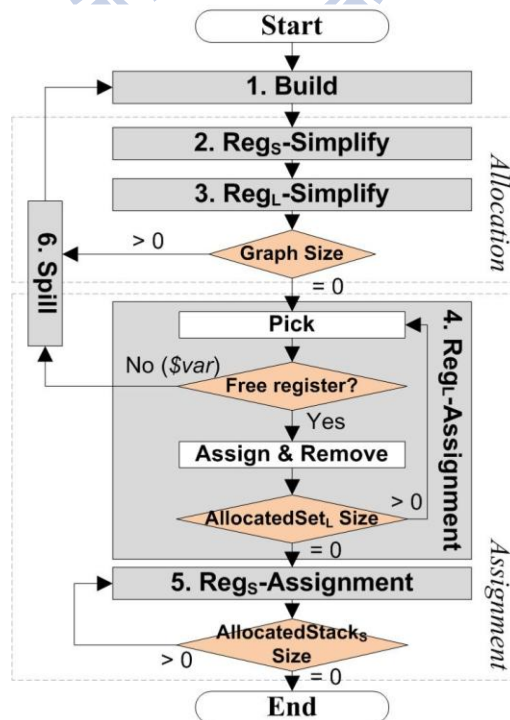


Figure 3-4 - Flowchart of the proposed algorithm based on Bottom-up graph coloring for mixed-width ISA with mode-switch by instruction encoding.

3.2.1 Allocation Pass

The main purpose of Allocation pass is to determine which of the variables should be in register or in memory at each execution point. As shown in Figure 3-4, allocation pass consists of three main stages, namely Reg_S-Simplify, Reg_L-Simplify, and Spill. Firstly, Reg_S-Simplify and Reg_L-Simplify stages remove some variables from the graph and place them into the corresponding stacks. If there are any remaining variables after these two stages, then the degrees of these variables will be all larger than RSN_A, and the algorithm will enter spill stage. Spill stage shrinks the degrees of variables by removing some variables with less effect on the execution performance from the graph. In this stage, a variable removed from the graph is the one that must be spilled into memory. Since spilling a variable will generate several new variables, the algorithm must return to the Build stage and rebuild the graph, and then the allocation pass will be repeated again. When all variables are removed from the graph, it will enter the assignment pass to assign physical registers to variables.

1896

A. Reg_S-Simplify and Reg_L-Simplify Stages

Reg_S-Simplify and Reg_L-Simplify stages are designed to remove variables from the graph. The Reg_S-Simplify stage is performed before the Reg_L-Simplify stage to find out variables which can guaranteed be assigned in $RegisterSet_S$. It will remove $\$Var_U$s with degree less than RSN_S, i.e., these variables can be assigned in $RegisterSet_S$ definitely. However, in Reg_L-Simplify stage, the removed target becomes the variables with degree less than or equal to RSN_A. This is because of the special case called "diamond" graph as shown in Figure 3-5. For example, if the target processor has two physical registers, r0 and r1, and the simplify stage removes only variables with degree less than two. Then the simplify stage removes no variables, and one of the

four variables, A, B, C, and D, will be chosen to spill. However, it is obvious that this diamond graph can be colored by two colors as shown in Figure 3-5. To tackle this problem, the proposed algorithm removes variables with degree less than or equal to RSN_A to make sure the remaining variables are those "must" be spilled.

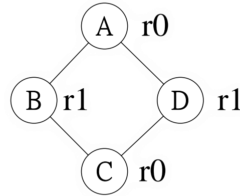


Figure 3-5 – Diamond graph.

For both Reg_S-Simplify and Reg_L-Simplify stages, the removing order is arranged from lower to higher degree; and after removing a variable from the graph, the degree of all remaining variables will be updated. The variables removed from Reg_S-Simplify are pushed into $AllocatedStacks_S$, and from Reg_L-Simplify stages are stored into $AllocatedSet_L$. Since the degrees of the variables removed in Reg_S-Simplify stage are smaller than RSN_S, all variables in $AllocatedStacks_S$ are able to be assigned in $RegisterSets_S$. As for variables in $AllocatedSet_L$, they may be assigned either in $RegisterSets_S$ or $RegisterSet_A$. Therefore, the way to assign proper variables to $RegisterSets_S$ is the key issue in our design as the red arrow as shown in Figure 3-6.

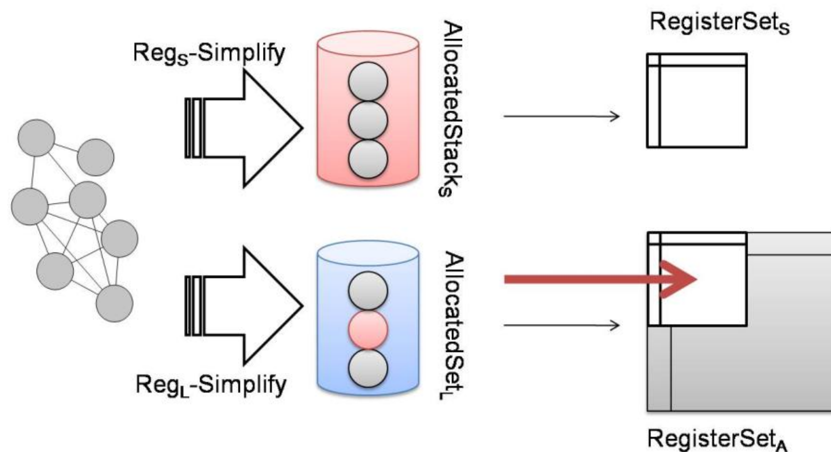
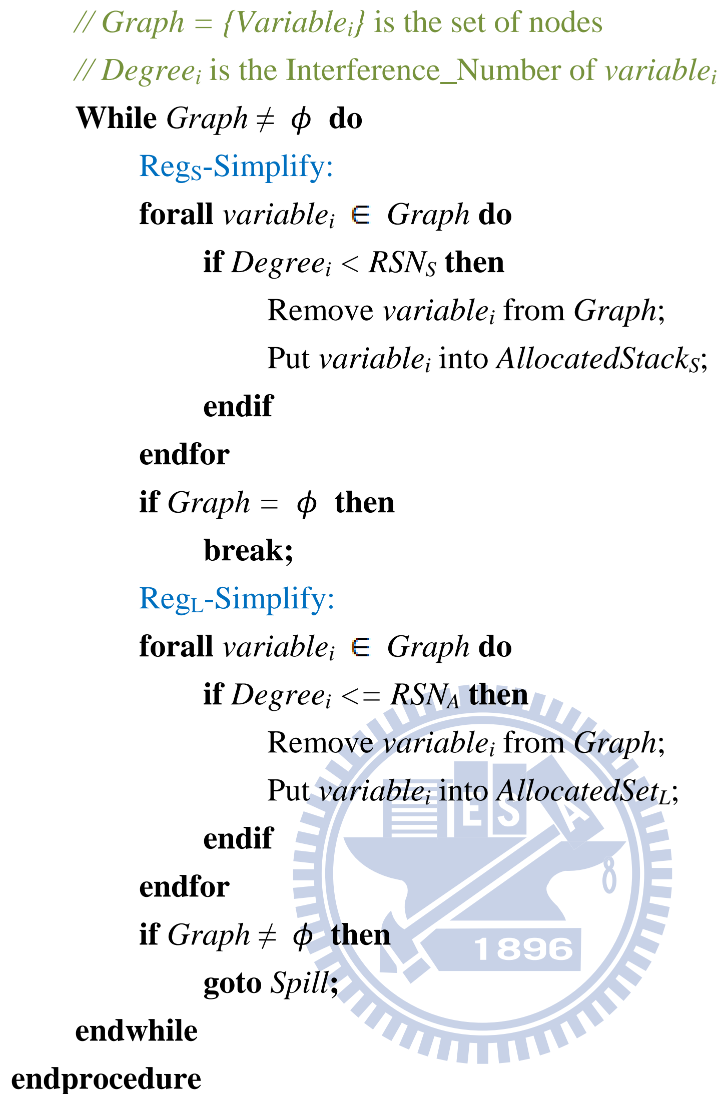


Figure 3-6 - The process sketch map of our proposed algorithm based on Bottom-up graph coloring.

Reg_S-Simplify and Reg_L-Simplify stages are performed until (1) none of the remaining variables in the graph can be removed; or (2) only variables with degrees equal to or larger than RSN_S in Reg_S-Simplify stage or larger than RSN_A in Reg_L-Simplify stage are left.

Note that all variables removed in Reg_L-Simplify stage are stored into $AllocatedSet_L$, even if their degrees are smaller than RSN_S. Since the degrees of variables may decrease over time, the variables removed later will have smaller degrees. After removing a certain number of variables, the degrees of some remaining variables in the graph will be less than RSN_S. Now these are two alternatives for choice: (1) let the algorithm go back to Reg_S-Simplify stage to remove these variables and put them into $AllocatedStacks_S$; or (2) stay in Reg_L-Simplify. If the algorithm can return to Reg_S-Simplify stage while in Reg_L-Simplify when there are variables with degrees less than RSN_S, then all of these variables will be pushed into $AllocatedStacks_S$ (i.e., be assigned in $RegisterSets_S$). However, these $\$Var_U$s may be not the proper variables to be assigned to $RegisterSets_S$ to increase the number of S-Format instructions than the other variables in $AllocatedSet_L$, and they may occupy registers from their neighbors which are more proper than these variables. Therefore, in order to avoid assigning improper variables in $RegisterSets_S$, all variables removed in Reg_L-Simplify stage are only stored into $AllocatedSet_L$ and let the assignment pass to discover which variables in $AllocatedSet_L$ should be assigned to $RegisterSets_S$. The pseudo code of allocation pass is shown in Figure 3-7.



```
Procedure Allocation
    // Graph = {Variable_i} is the set of nodes
    // Degree_i is the Interference_Number of variable_i
    While Graph ≠ ϕ do
        Reg_S-Simplify:
        forall variable_i ∈ Graph do
            if Degree_i < RSN_S then
                Remove variable_i from Graph;
                Put variable_i into AllocatedStack_S;
            endif
        endfor
        if Graph = ϕ then
            break;
        Reg_L-Simplify:
        forall variable_i ∈ Graph do
            if Degree_i <= RSN_A then
                Remove variable_i from Graph;
                Put variable_i into AllocatedSet_L;
            endif
        endfor
        if Graph ≠ ϕ then
            goto Spill;
    endwhile
endprocedure
```

Figure 3-7 - Pseudo code of allocation pass.

B. Spill Stage

In Reg_S-Simplify and Reg_L-Simplify stages, we remove variables from the graph. If there are remaining variables in the graph and cannot be removed anymore, the algorithm will enter the spill stage. Then spill stage must choose one of the remaining variables to spill, i.e. insert spill codes for this variable. An ideal variable for spilling is the one that requires less number of dynamic loads and stores (less number of

accesses) and can reduce the number of future potential spill variables (higher degree). Accordingly, the spill cost of $variable_i$, $SpillCost_i$, is defined as follows:

$$SpillCost_i = \frac{Access_i}{Degree_i},$$

where $Access_i$ is the number of defines and uses of $variable_i$, and $Degree_i$ is the number of edges in the interference graph connected to $variable_i$. After choosing $variable_i$, spill stage will insert corresponding load/store instructions for it, and then return to the Build stage. Since inserting a spill code will result in new temporary variables, it is necessary to rebuild the interference graph. In other words, the entire algorithm will be repeated again if a new actual spill is generated.

3.2.2 Assignment Pass

Assignment pass assigns a physical register to each variable in $AllocatedStacks_S$ and $AllocatedSet_L$. This pass is divided into two stages, namely Reg_L-Assignment and Reg_S-Assignment. The Reg_L-Assignment stage is performed before the Reg_S-Assignment stage because that variables in $AllocatedStacks_S$ can be guaranteed they have registers in $RegisterSets_S$ to use. As for more details, we will discuss in Section 3.2.3.

For the variables in $AllocatedSet_L$, they may be assigned in $RegisterSet_A$ or in $RegisterSets_S$. That is, Reg_L-Assignment stage must determine the assignment target, $RegisterSets_S$ or $RegisterSet_A$, for the variables rather than just assigns all of them into $RegisterSet_A$. The assignment target determination can be viewed as the problem of deciding which variables in $AllocatedSet_L$ should be assigned in $RegisterSets_S$. The code size reduction is proportional to the number of U-INSs being translated to S-Format instructions, and the translation ratio is dominated by the number of $\$Var_U$s being

assigned in *$RegisterSets_S$*. Due to the size limitation of *$RegisterSets_S$*, not all *$\$Var_U$*s can be assigned into *$RegisterSets_S$*. To determine which *$\$Var_U$* should be assigned into *$RegisterSets_S$*, we define a profit function in this paper. The profit function considers two features of *$\$Var_U$*, namely utilization and degree.

The utilization of *$\$Var_U$* is the number of times that *$\$Var_U$* is being accessed by S-Format instructions in a whole function. Obviously, a frequently accessed *$\$Var_U$* should have higher priority to be assigned to *$RegisterSets_S$*. However, it is impossible to calculate the actual number of accesses by S-Format instructions of a *$\$Var_U$* before the instruction formatting pass in which the final instruction format has been decided for INSs, and, as mentioned above, the instruction formatting pass can only be performed after register allocation and assignment. To overcome this problem, we propose an approach to evaluate the approximate number of accesses of a *$\$Var_U$*. The approach consists of two parts, namely static and adaptive estimations. The static estimation calculates the initial number of times that a *$\$Var_U$* is being accessed by U-INSs. Note that this value is a constant for each *$\$Var_U$*. As for the adaptive estimation, it computes the current number of times that a *$\$Var_U$* is being accessed by U-INSs which have at least one operand being assigned into *$RegisterSets_S$*. The adaptive estimation of all *$\$Var_U$*s must be updated whenever a *$\$Var_U$* is assigned to *$RegisterSets_S$*.

The degree of *$\$Var_U$* represents the number of neighboring variables of *$\$Var_U$* in the graph. In order to make more *$\$Var_U$* be assigned in *$RegisterSets_S$*, a *$\$Var_U$* with less degree should have higher opportunity to be assigned into *$RegisterSets_S$*. Note that less degree also implies shorter live range. Based on these two features, the profit value for *$\$Var_{Ui}$*, denoted as *$Sel_profit_i$*, is defined as follows:

$$Sel_profit_i = \frac{\alpha \times Init_est_i + (1-\alpha) \times Adp_est_i}{Degree_i + 1} ,$$

where $Init_est_i$ and Adp_est_i represent the initial and adaptive estimation results for $\$Var_{Ui}$, respectively; $Degree_i$ is the degree of $\$Var_{Ui}$. Because of the continuous changing of Adp_est_i, Sel_profit_i also has to be updated after a $\$Var_U$ is assigned to $RegisterSets_S$. Note that alpha value, α , in Sel_profit_i is the weight parameter between $Init_est_i$ and Adp_est_i, and we will experiment this algorithm with different alpha values in Chapter 4.

Moreover, during Reg_L-Assignment stage it is possible to encounter a variable which cannot be assigned into any physical registers, i.e., all available physical registers are occupied by its neighbors. Then the algorithm will enter the spill stage and insert spill codes for this variable directly, and the entire algorithm will be repeated again. Remind that, in this algorithm, variables which definitely cannot be assigned to any register have been discovered at allocation pass. For those variables which may not possibly be assigned to registers, they will be discovered in Reg_L-Assignment stage.

As mentioned above, the variables allocated into $AllocatedStack_S$ are guaranteed to be assigned in $RegisterSets_S$, since their degrees are smaller than RSN_S. Accordingly, Reg_S-Assignment stage may adopt the traditional assignment approach by following the stack pop order to choose the physical registers for the variables in $AllocatedStack_S$.

The pseudo code of assignment pass is shown in Figure 3-8.

```
Procedure Assignment
    While AllocatedSet_L ≠ ϕ do
        Forall variable_i ∈ AllocatedSet_L do
            Calculate Sel_Profit_i;
        endfor
        pick one variable_i with highest Sel_Profit_i as variable*;
        if variable* can be assigned into RegisterSet_S then
            Assign a register different from its neighbors to variable* in RegisterSet_S;
            Remove variable* from AllocatedSet_L;
        else if variable* can be assigned into RegisterSet_A then
            Assign a register different from its neighbors to variable* in RegisterSet_A;
            Remove variable*from AllocatedSet_L;
        else
            /* it has no register to assign, pass it to spill */
            goto Spill(variable*);
        endif
    endwhile
    Forall variable_i ∈ AllocatedStacks_S do
        Assign a register different from its neighbors to variable_i by stack pop order;
    endfor
endprocedure
```

Figure 3-8 - Pseudo code of assignment pass.

3.2.3 Discussion

In this subsection, three issues will be discussed: (1) the effect of different assignment orders, (2) assignment without Reg_S-Simplify, and (3) extension of the algorithm to more hierarchy register sets for different S-Formats.

A. Different Assignment Orders in Assignment

In the proposed algorithm based on bottom-up graph coloring, the Reg_L-Assignment stage is processed before the Reg_S-Assignment stage. The main reason is that variables in $AllocatedStacks_S$ are guaranteed that they have registers in

$RegisterSet_S$ to use. In other words, no matter how the Reg_L-Assignment assigns the variables in $AllocatedSet_L$, the Reg_S-Assignment can assign registers in $RegisterSet_S$ to variables in $AllocatedStack_S$. Since only variables with degree less than RSN_S are pushed into $AllocatedStack_S$ during Reg_S-Simplify. For example, there is an interference graph as shown in Figure 3-9. Assume that the architecture has three registers, \$r0, \$r1, and \$r2. $RegisterSet_S$ contains \$r0 and \$r1 (RSN_S = 2), and $RegisterSet_A$ contains all of the three registers (RSN_A = 3). Variables D and E in Figure 3-9 are simplified by Reg_S-Simplify and moved to $AllocatedStack_S$ since their degrees are less than two. The others variables A, B, and C are simplified by Reg_L-Simplify and moved to $AllocatedSet_L$. The sel_profit_i for variable A, B, and C in Reg_L-Assignment is A > B > C. As the result is in Figure 3-9 (a), if we assign $AllocatedSet_L$ first, variable A can be assigned to \$r0 because that there are no registers have been occupied, then variable B is assigned in \$r1, and the last register \$r2 is assigned to variable C. After performing Reg_L-Assignment, Reg_S-Assignment is invoked to handle $AllocatedStack_S$. Since variables in $AllocatedStack_S$ are guaranteed to have registers in $RegisterSet_S$ to use, we just assign registers which are not occupied by their neighbors in $RegisterSet_S$. For the result shown in Figure 3-9 (a), all variables in $AllocatedStack_S$ are assigned to $RegisterSet_S$, and it is what we expect that variable A and B should have higher opportunity to be assigned to $RegisterSet_S$ than variable C does.

However, if we change the assignment order and let Reg_S-Assignment be performed before the Reg_L-Assignment, it may result in that some of the variables in $AllocatedSet_L$ with higher priority which may be assigned to $RegisterSet_S$ in the previous order cannot be assigned to $RegisterSet_S$ now. For example, if the assignment order is Reg_S-Assignment $\rightarrow$ Reg_L-Assignment as shown in Figure 3-9 (b), Variables

D and E in $AllocatedStacks_S$ will be assigned to \$r0 directly first. Then, variables A, B, and C in $AllocatedSet_L$ will be picked up by the sel_profit_i value in descendant. First, variable A is picked up and assigned to \$r1 (\$r0 is occupied by its neighbor), then variable B is assigned to \$r2, and the last variable C is assigned to \$r0. Because the $sel_profit_B > sel_profit_C$, we desire that chance of variable B in $RegisterSets_S$ is higher than variable C. However, variable B is not assigned to $RegisterSets_S$, but the variable C does.

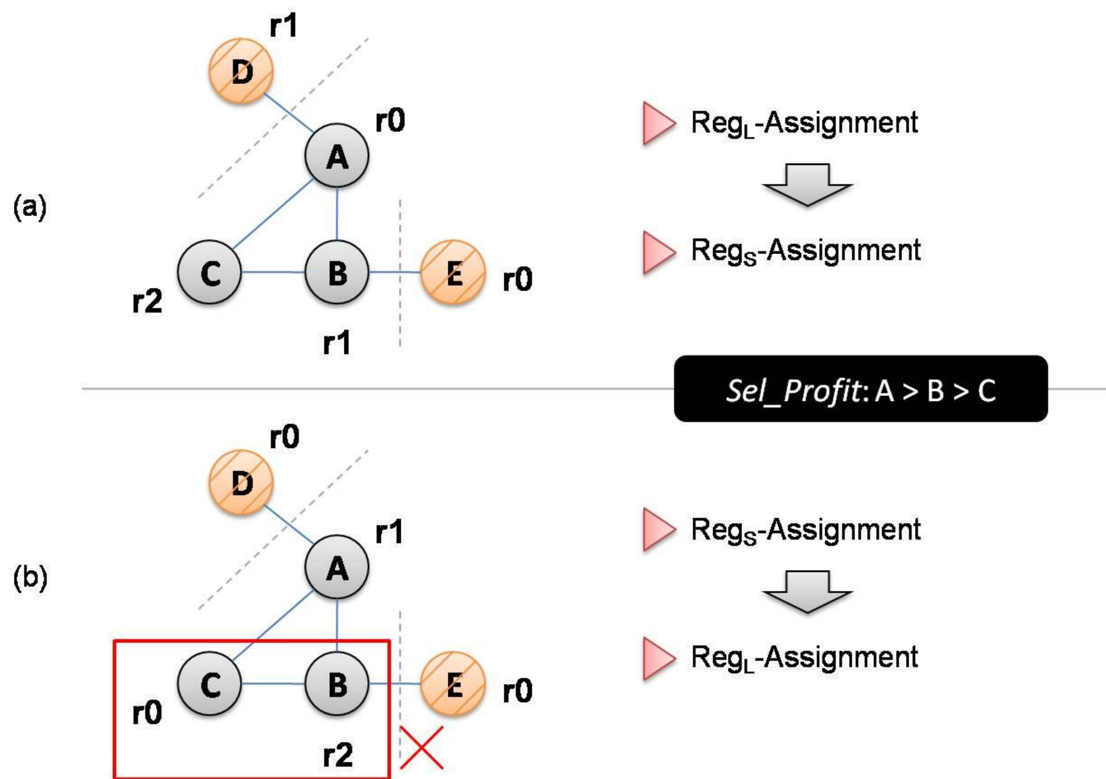


Figure 3-9 - Different Assignment Order in Bottom-up Graph Coloring for Mixed-width ISA with mode switch by instruction encoding : (a) Reg_L-Assignment $\rightarrow$ Reg_S-Assignment (b) Reg_S-Assignment $\rightarrow$ Reg_L-Assignment.

B. Assignment without Reg_S-Simplify

The second discussion issue is how about if we do not isolate variables with degree less than RSN_S in Reg_S-Simplify of allocation pass, i.e., let all variables be simplified to $AllocatedSet_L$ in Allocation pass, and then assign those variables in descending order of *sel_profit* values in Assignment pass. This method seems instinctive. However, if we do not isolate variables in Reg_S-Simplify, the *sel_profit* of some variables which are simplified to $AllocatedStacks_S$ originally may be higher than those which are simplified to $AllocatedSet_L$ originally. Therefore, it will cause the problem similar to that of the assignment order "Reg_S-Assignment → Reg_L-Assignment" mentioned above.

For the same example given above, Figure 3-10 is the result of register assignment if Reg_S-Simplify stage is canceled in the allocation pass. Because all non-spilled variables are in $AllocatedSet_L$ and all of them are given a *sel_profit* as their priority, in this example, the priority is D > A > E > B > C. According to this assignment order, the results in that variable B cannot be assigned in $RegisterSets_S$ since variable E have higher priority than variable B.

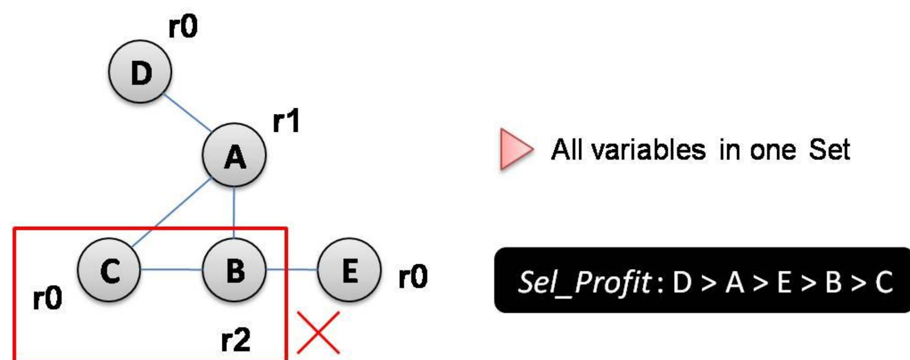


Figure 3-10 - Assignment result without Reg_S-Simplify.

C. Extension of the Algorithm to More Hierarchy Register Sets for Different S-Formats

For extending algorithm to handle the 1R4 form, we can modify the algorithm as shown in Figure 3-11. Here we have a new register class named *$RegisterSet_M$*, and it contains registers (eg. \$r1 ~ \$r15) can be accessed by those instructions in 1R4 form, i.e. the *RSN_M* is 15. Moreover, the Reg_M-Simplify and Reg_M-Assignment stages are added in the extended algorithm, and the U-INS may contain 1R4, 3R3, 2R3, and 1R3 forms. For \$$Var_U$s in U-INS, a \$$Var_U$ which is defined or used only by 1R4 form instructions is denoted as \$$Var_{U4}$, and the others are denoted as \$$Var_{U3}$.

At the Reg_M-Simplify, the algorithm removes \$$Var_{U4}$s which are with degree less than *RSN_M* to *$AllocatedStack_M$*. In other words, if a variable is defined/used only by 1R4 instructions and the variable can be assigned to *$RegisterSet_M$* definitely, and we directly push it into *$AllocatedStack_M$* and assign it to *$RegisterSet_M$* at Reg_M-Assignment stage. As for those \$$Var_{U4}$s with the degrees are larger than or equal to *RSN_M*, they may be removed at the Reg_L-Simplify stage and stored into *$AllocatedSet_L$*. Therefore, the modification of the *sel_profit* function is needed. The modified *sel_profit* function should take those \$$Var_{U4}$s into account. For each \$$Var_{U4}$, it increases the *Init_est* and *Adp_est* in *sel_profit*, but the weight should be lower than \$$Var_{U3}$. Since \$$Var_{U3}$s have higher register pressure than \$$Var_{U4}$s, they are intended to be assigned to *$RegisterSet_S$*. By adopting these modifications, the \$$Var_{U3}$s will still have the highest priority to use *$RegisterSet_S$*, then the second priority is the *$RegisterSet_M$* for \$$Var_{U4}$s, and the last is those variables that are not defined or used by U-INS.

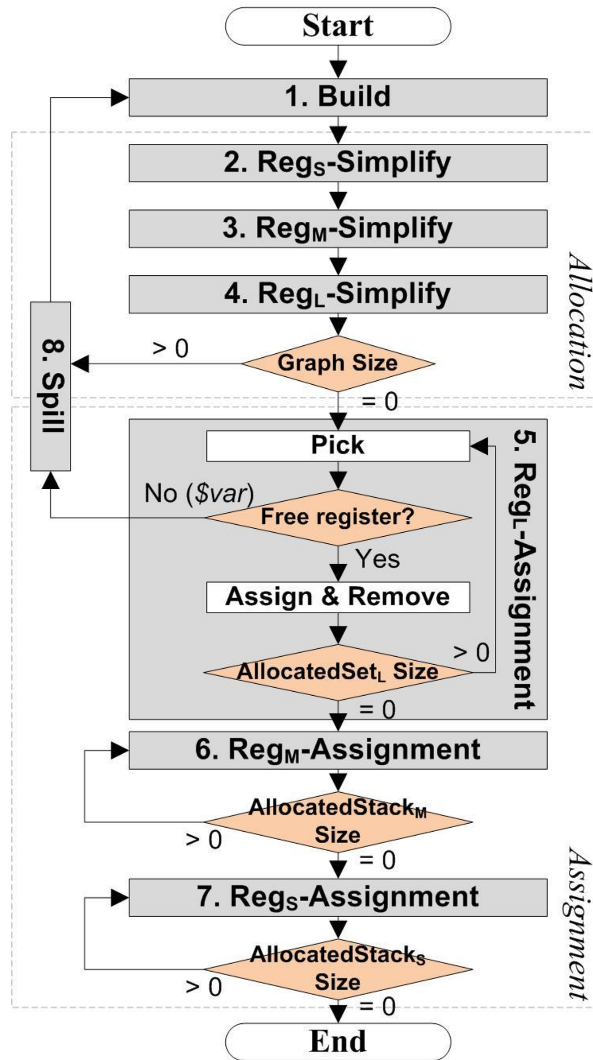


Figure 3-11 - Extension of Design I for different S-Formats.

3.3 Design II : Based on Priority-based Graph Coloring

The second design in this thesis is based on priority-based graph coloring. The mainly difference from the previous design is that this algorithm makes efforts on selecting variables to increase the number of S-Format instructions first, and then considers the number of spill codes. In opposite, the previous design discussed in Section 3.2 makes efforts on minimizing spill codes in allocation pass first, and then considers to increase the number of S-Format instructions in assignment pass.

For the proposed algorithm based on Priority-based graph coloring for mixed-width ISA with mode-switch by instruction encoding, its flowchart is shown in Figure 3-12. There are three sets of live range (live range is the same with variable) need to be allocated and assigned are defined as follows:

1. **LR_S:** LR_S (Live Ranges in $RegisterSet_S$) contains variables with interference number less than RSN_S. For those variables in LR_S, they are guaranteed to be allocated and assigned to $RegisterSet_S$.
2. **LR_U:** LR_U (Live Ranges in Uncertain RegisterSet) contains variables with interference number larger than or equal to RSN_S. For those variables in LR_U, they may be allocated and assigned to $RegisterSet_S$, $RegisterSet_A$, or Memory (Spilled). The way to determine which variables in LR_U should be assigned to $RegisterSet_S$ is critical to this algorithm.
3. **LR_L:** LR_L (Live Ranges in $RegisterSet_A$) contains variables which cannot be assigned a register in $RegisterSet_S$ during the LR_U Allocation and Assignment pass. They will be allocated and assigned to $RegisterSet_A$ or Memory (Spilled) during the LR_L Allocation and Assignment pass.

First, Build stage parses all necessary information (such as live rages, instruction types) and constructs the interference graph the same way as that described in the previous design. Then the Separate pass separates variables in the graph into LR_U and LR_S. After Separate stage, the algorithm uses a priority function which is designed to make more instructions to be encoded as S-Format instructions to choose variables from LR_U to allocate and assign registers in the Prioritize stage. During this pass, some variables in LR_U may be transferred to LR_L. For minimizing the generated spill codes, the algorithm uses the other priority function which is designed for minimizing the spill codes to allocate and assign registers for variables in LR_L. And finally, the variables in LR_S are handled. The input and output of the proposed algorithm are INSs with virtual registers and assembly code, respectively.

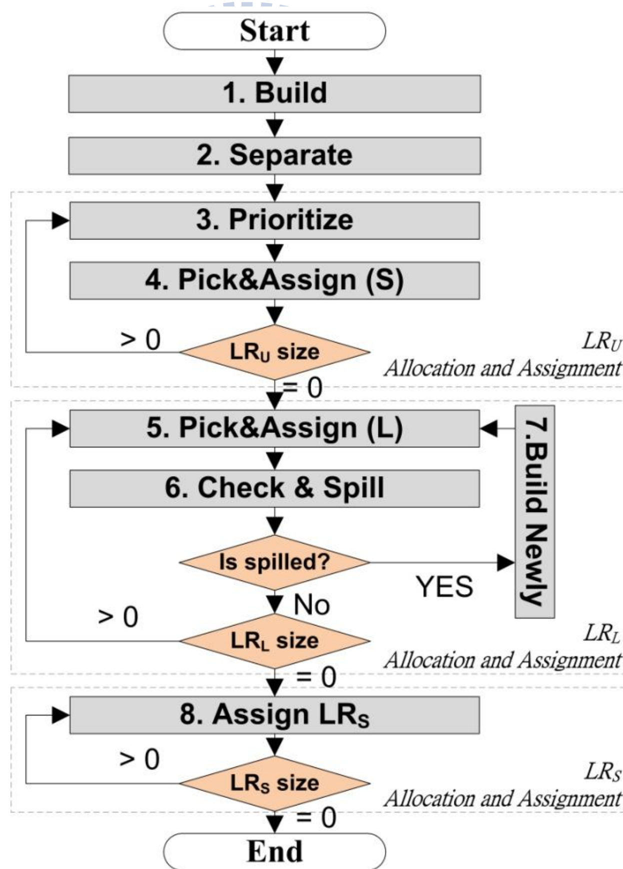


Figure 3-12 - Flowchart of the register allocation and assignment algorithm based on Priority-based graph coloring for mixed-width ISA with mode-switch by instruction encoding.

The pseudo code of Priority-based graph coloring for mixed-width ISA with mode-switch by instruction encoding is shown in Figure 3-13.

```
Procedure MxPrGCRA
    Build interference graph ifGraph;
    Separate LR_U and LR_S;
    While LR_U ≠ ϕ do
        Forall variable_i ∈ LR_U do
            Calculate Sel_Profit_i;
        endfor
        pick one variable_i with highest Sel_Profit_i;
        if variable_i can be assigned into RegisterSets_S then
            Assign a register different from its neighbors to variable_i in RegisterSets_S;
            Remove variable_i from LR_U;
        else
            Move variable_i to LR_L;
        endif
    endwhile
    While LR_L ≠ ϕ do
        pick one variable_i with highest SpillCost_i;
        Assign a register different from its neighbors to variable_i in RegisterSet_A;
        Remove variable_i from LR_L;
        Forall variable_i* ∈ neighbors of variable_i do
            if variable_i* has to be spilled then
                goto Spill(variable*);
                update ifGraph with new variables produced by spilling;
            endif
        endfor
    endwhile
    While LR_S ≠ ϕ do
        Forall variable_i ∈ LR_S do
            Assign a register in RegisterSets_S different from its neighbors to variable_i
        endfor
    endwhile
```

Figure 3-13 - Pseudo code of Priority-based graph coloring for mixed-width ISA with mode-switch by instruction encoding

3.3.1 *Separate Stage*

After the interference graph has been built by Build stage, Separate stage separates variables in the graph into LR_U and LR_S by their interference number. Note that the variables in LR_L are not discovered in Separate stage because that variables in LR_U may be allocated into *RegisterSet*$_S$, *RegisterSet*$_A$, or Memory (Spilled), and thus, variables in LR_L are included in LR_U currently.

3.3.2 *LR_U Allocation and Assignment*

During the LR_U Allocation and Assignment pass, the algorithm needs to find out which variables in LR_U should be allocated and assigned to *RegsiterSet*$_S$. Because that the variables in LR_U are those variables with interference number larger than or equal to *RSN*$_S$, it means that only a part of them can be assigned to *RegsiterSet*$_S$ and the others will be assigned to either *RegisterSet*$_A$ or Memory. Therefore, the way to determine which variables in LR_U should be assigned to *RegisterSet*$_S$ is critical as mentioned above. To achieve this, a priority function for prioritize stage is used. The Prioritize stage calculates priority for each variable in LR_U to make a proper order to assign registers, and the higher priority (profit) variable has higher chance to get a register in *RegisterSet*$_S$. The priority (profit) value for *\$Var*$_{Ui}$ (denoted as *Sel_profit*$_i$) is the same as that defined in the previous design and listed as follows:

$$Sel_profit_n = \frac{\alpha \times Init_est_n + (1-\alpha) \times Adp_est_n}{Degree_n + 1}$$

When all variables in LR_U have been given a priority, Pick&Assign(S) stage picks the highest priority variable *variable*$_i$, and try to assign a register in *RegisterSet*$_S$ to it. If there is a free register in *RegisterSet*$_S$ which is not occupied by the neighbors of *variable*$_i$,

then $variable_i$ will be assigned this register and removed from LR_U. Otherwise, $variable_i$ will be moved to LR_L.

After assigning a variable into $RegisterSet_S$, this pass will check the LR_U. If it is empty, then the algorithm will go to the next pass - "LR_L Allocation and Assignment". Otherwise, it will repeat the Prioritize stage to update the *Adp_est* for the remaining variables in LR_U and the Pick&Assign(S) stage to tackle the remaining variables in LR_U until LR_U is empty.

3.3.3 LR_L Allocation and Assignment

In the previous pass, variables in LR_U those cannot be directly assigned to $RegisterSet_S$ have been found and moved to LR_L. The goal of this pass is to determine which variables in LR_L should be assigned to $RegisterSet_A$ and the others should be spilled to memory. Apparently, the allocation and assignment for LR_L is critical to reduce the number of generated spill codes.

To minimizing the generated spill codes, a priority function for evaluating the spill cost of a variable is evolved. First, the algorithm assumes that all variables in LR_L are initially stored in memory, and uses the priority function to estimate how many processor cycles can be saved if $variable_i$ in LR_L is assigned a physical register. For those variables which can save more cycles, they should have higher priority to get registers than others. This priority function is similar to the traditional priority-based graph coloring for register allocation and listed as follows:

$$SpillCost_n = \frac{SaveCycle_n}{Degree_n},$$

where $SaveCycle_i$ represents the number of processor cycles saved if $variable_i$ is assigned a physical register rather than stored in memory, and $SaveCycle_i$ is usually

proportional to the access counts of $variable_i$; and $Degree_i$ is the number of edges connected to $variable_i$ in the interference graph. This priority function makes the variable with higher $SaveCycle_i$ and lower $Degree_i$ have higher priority to get a physical register in $RegisterSet_A$. Pick&Assign(L) stage picks the highest priority (*SpillCost*) $variable_i$ and assigns a register in $RegisterSet_A$ to it. Then the algorithm goes through each neighbor of $variable_i$ to check whether it needs to be spilled or not. If there is a variable spilled, the corresponding load/store instructions for spilled variable will be inserted. Then Build Newly stage will build new variables and move them into LR_L for the new temporary variables produced during spilling. Otherwise, the algorithm will go back to Pick&Assign(L) stage when LR_L is not empty or go to the next pass – "LR_S Allocation and Assignment" when LR_L is empty.

3.3.4 LR_S Allocation and Assignment

Since all variables in LR_S have degree less than RSN_S, i.e., the simultaneously alive variables with each variable in LR_S of a function are no more than RSN_S, all of them can be assigned to $RegisterSet_S$ without any constrains. Therefore, the LR_S Allocation and Assignment pass just picks up a variable in LR_S and assigns a register in $RegisterSet_S$ to it until LR_S is empty.

3.3.5 Discussion

In this subsection, an alternative design without the "LR_U allocation and assignment pass" and the extension of the algorithm to more hierarchy register sets for different S-Formats will be discussed.

A. Alternative Design

The traditional priority-based graph coloring uses only one priority function to pick up variables to assign. In order to increase the number of instructions encoded as S-Format instruction, we can directly use *Sel_profit* function to instead the traditional priority function as shown in Figure 3-14. Consequently, the Pick&Assign stage will pick the variable with highest *Sel_profit* up and assign a register to the variable, and thus, increasing the number of S-Format instructions. However, the algorithm is not good enough because the influence of spill codes is not taken into account. Therefore, we should allocate and assign registers hierarchically while using two priority functions to increase the S-Format translation rate and decrease the number of spill codes, respectively.

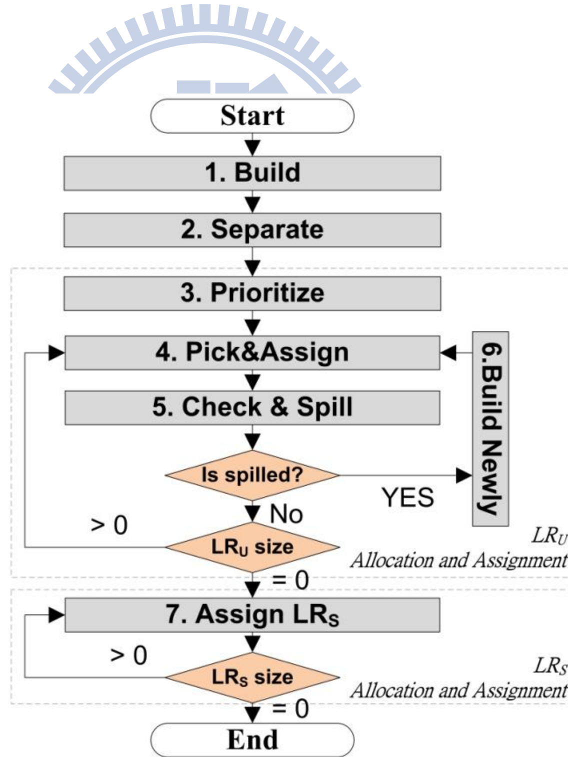


Figure 3-14 - Flowchart of the proposed algorithm without LR_L allocation and assignment pass of Design II.

This is why we move a variable which cannot be assigned in $RegisterSets_S$ to LR_L in "LR_U allocation and assignment pass" (as shown in Figure 3-12). Because this

variable cannot use registers in *RegisterSet*$_S$, it only can either be allocated in *RegisterSet*$_A$ or be spilled. Therefore, this variable should not use the *Sel_profit* priority function to determine its priority because the *Sel_profit* priority function is not designed for minimizing the number of spill codes. Hence, the priority function which is designed to determine the priority to use *RegisterSet*$_A$ or to be spilled should be adapted instead here. So we move these variables from LR_U to LR_L, and they will be handled in "LR_L allocation and assignment pass" pass. The evaluating of the proposed algorithm with and without "LR_L allocation and assignment pass" will be shown in Chapter 4.

B. Extension of the Algorithm to More Hierarchy Register Sets for Different S-Formats

Similar with Section 3.2.3, the proposed algorithm based on priority-based graph coloring can be extended to adopt different S-Formats. To achieve this, we use hierarchical allocation and assignment. First we find out which variables should be in *RegisterSet*$_S$, and then we find out which variables should be in *RegisterSet*$_M$, the last, considering the spill codes for the variables allocated in *RegsiterSet*$_A$.

The modified algorithm based on priority-based graph coloring is shown in Figure 3-15. It is obviously that the "LR_M Allocation and Assignment pass" is added. During "LR_U Allocation and Assignment pass", variables those cannot be assigned in *RegisterSet*$_S$ will be moved to LR_M rather than LR_L. Then, a new priority function, *Priority_M*$_i$, is evolved for each *variable*$_i$ in Prioritize(M) stage of "LR_M Allocation

and Assignment pass". This priority function makes $Var_{U4}s have higher priority. Therefore, the Pick&Assign(M) stage picks variables with higher *Priority_M* up and try to assign a register in $RegisterSet_M$ to this variable. If there is no free registers in $RegisterSet_M$, and then the algorithm moves this variable into LR_L.

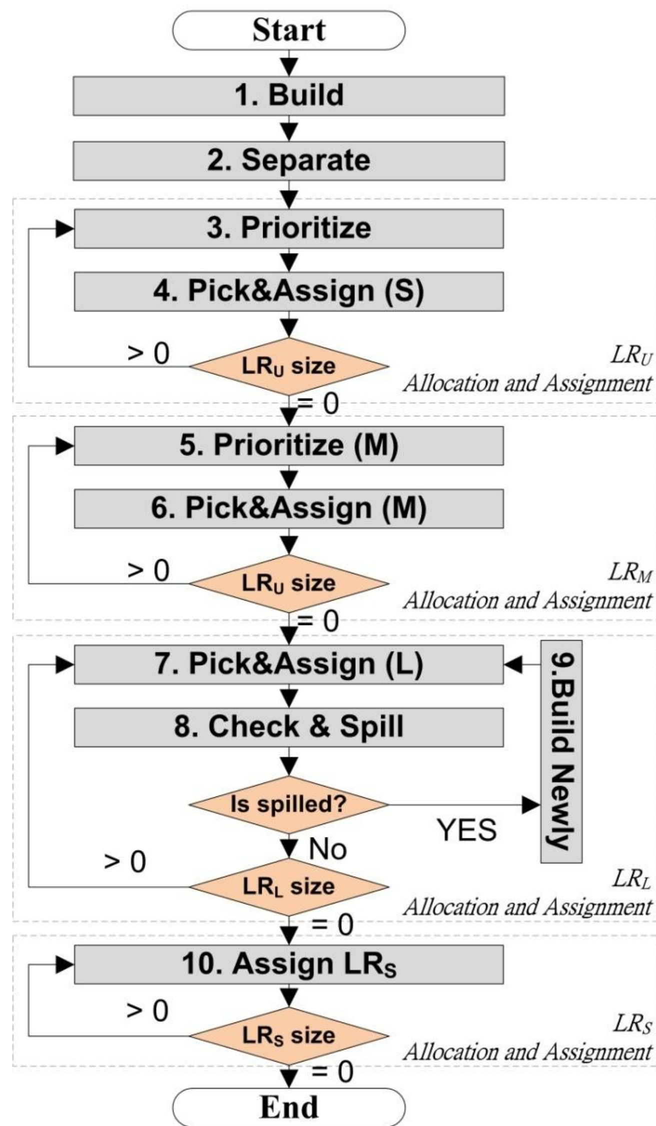


Figure 3-15 - Extension of Design II for different S-Formats.

Chapter 4 Experiment

In this chapter, the experiment environment and the simulation result are described. First the experimentation environment is introduced. The next, we will determine the parameter, α value. And finally, benchmark evaluation results for two algorithms proposed and design alternatives discussed in Section 3.2.3 and 3.3.5 will be shown out also. All of these evaluation results contain the code size reduction rate and the spill code percentage.

4.1 Environment

The Low-Level Virtual Machine (LLVM) is used as our compiler infrastructure [12][13]. It provides back-ends for lots of popular architectures, for examples, ARM, MIPS, x86, PowerPC, etc. LLVM back-end has many useful passes such as coalescing, instruction selection, mid-level optimizers, etc. To generate INSs with virtual registers, we modified the LLVM compiler back-end to produce the INSs with virtual registers as input files of our algorithm. The modified LLVM back-end will ignore the physical registers described in target machine description file (.td) and treat the number of available physical registers as infinite. In addition, LLVM back-end uses a constant pool to hold large constants, and thus, the limitation by the fewer bits to hold immediate values can be neglected.

In our simulation, the target ISA is MIPS/MIPS16 with assumption that instruction mode (S-Formant or L-Format) is changed by a specified instruction bit rather than the mode switching instruction. According to the register file of MIPS, $r0 is used as zero, $r26 and $r27 are reserved by kernel usage, and $GP, $SP, $FP, and $RA as special registers, so they cannot be used in other way. Therefore, the parameters of registers are assumed as following: $RegisterSets_S$ = $r1 ~ $r7, $RSN_S = 7$, $RegisterSet_A$ = $r1 ~ $r25, , and $RSN_A = 25$. In addition,

benchmarks used in this experiment were selected from SPECINT2000, Mibench, and Mediabench.

4.2 Benchmark Evaluation Results

In this section, α value in the *sel_profit* function is determined and our simulation results including code size reduction, spill codes, and the S-Format limitations analysis of the generated codes are presented. The simulation result of the benchmark programs for our proposed algorithms are denoted as the MxBuGCRA (named by Bottom-up Graph Coloring Register Allocation for Mixed-width ISA) and MxPrGCRA (named by Priority-based Graph Coloring Register Allocation for Mixed-width ISA). For comparison, the simulation results of the traditional bottom-up graph color register allocation (BuGCRA) and priority-based graph coloring register allocation (PrGCRA) are also depicted.

4.2.1 Parameter Determination

The α value is the parameter to control the weight between $Init_est_i$ and Adp_est_i in our heuristic function, the sel_profit_i, which is used in "Reg_L-Assignment Pass of MxBuGCRA" and "LR_U Allocation and Assignment pass of MxPrGCRA". If α is equal to one, it means that the sel_profit_i takes only the $Init_est_i$ into account, and if α is equal to zero, the sel_profit_i takes only the Adp_est_i into account, apparently.

We have evaluated α value from 0.01 to 1 for two algorithms proposed in this thesis, MxBuGCRA and MxPrGCRA, and the result is shown in Figure 4-1. Note that we do not let α value be equal to zero because that the Adp_est_i of all variables are zero initially, and sel_profit_i for all variables will also be zero once α value is zero. It

results in that the first variable is selected by random, and the selected variable may have no or little benefit to increase the number of S-Format instructions.

From the evaluation results, we observe that whenever α is approaching zero or approaching one, the code size reduction is decreasing. It is reasonable that we should not emphasize too much either $Init_est_i$ or Adp_est_i. If the $Init_est_i$ is over emphasized, the *sel_profit* function cannot increase the appropriate priority for the other variables in U-INSs that access a variable which had been just assigned in $RegisterSets_S$. If the Adp_est_i is over emphasized, the heuristic approach may sink in the local optimal result easily. In addition, the changes of the percentages of spill codes seem irregular for different α values. Therefore, for the code size reduction objective, α value may be set to 0.4 for the proposed algorithms.

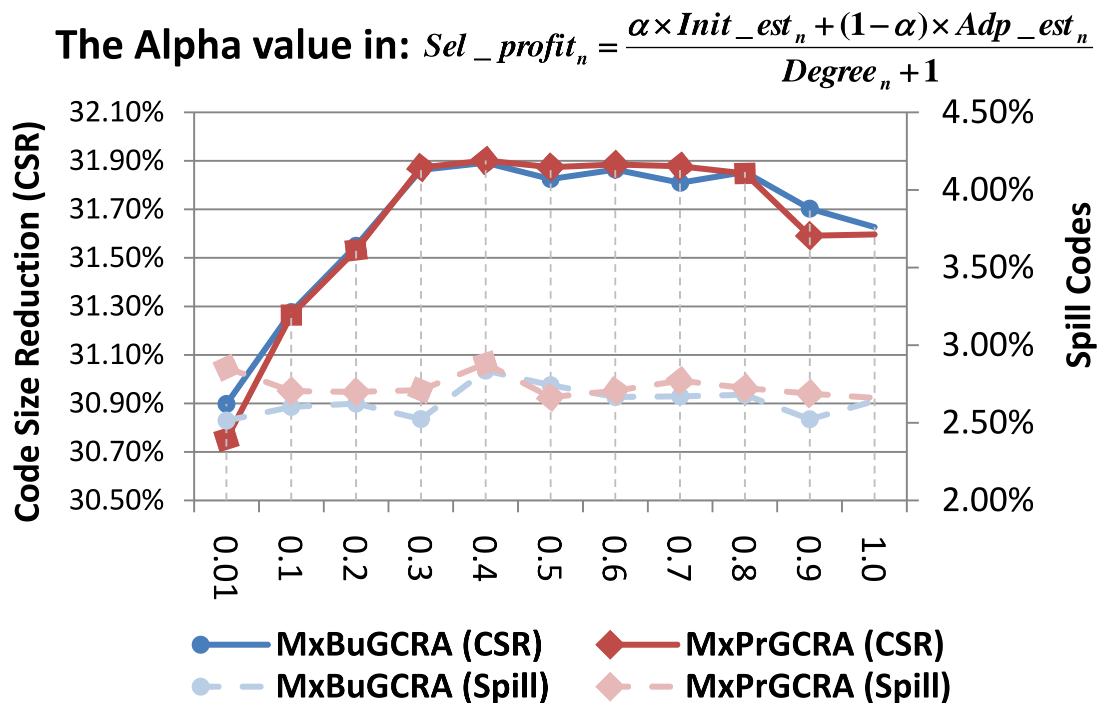


Figure 4-1 - The evaluation of differentα values.

4.2.2 Comparisons of Design Alternatives

The experiment results for the different approaches discussed in section 3.2.3 and 3.3.5 are presented as follows. Note that we have simulated different α values for each approach, and all of the approaches have the best results when α is 0.4 as the same as the proposed algorithms.

The first one is the different assignment order of the Assignment pass in MxBuGCRA. As the same with the examples given in section 3.2.3, the assignment order "Reg_L-Assignment → Reg_S-Assignment" is better than the assignment order "Reg_S-Assignment → Reg_L-Assignment" and "without isolating $AllocatedStacks_S$". Also, the results of these alternatives are shown in Figure 4-2.

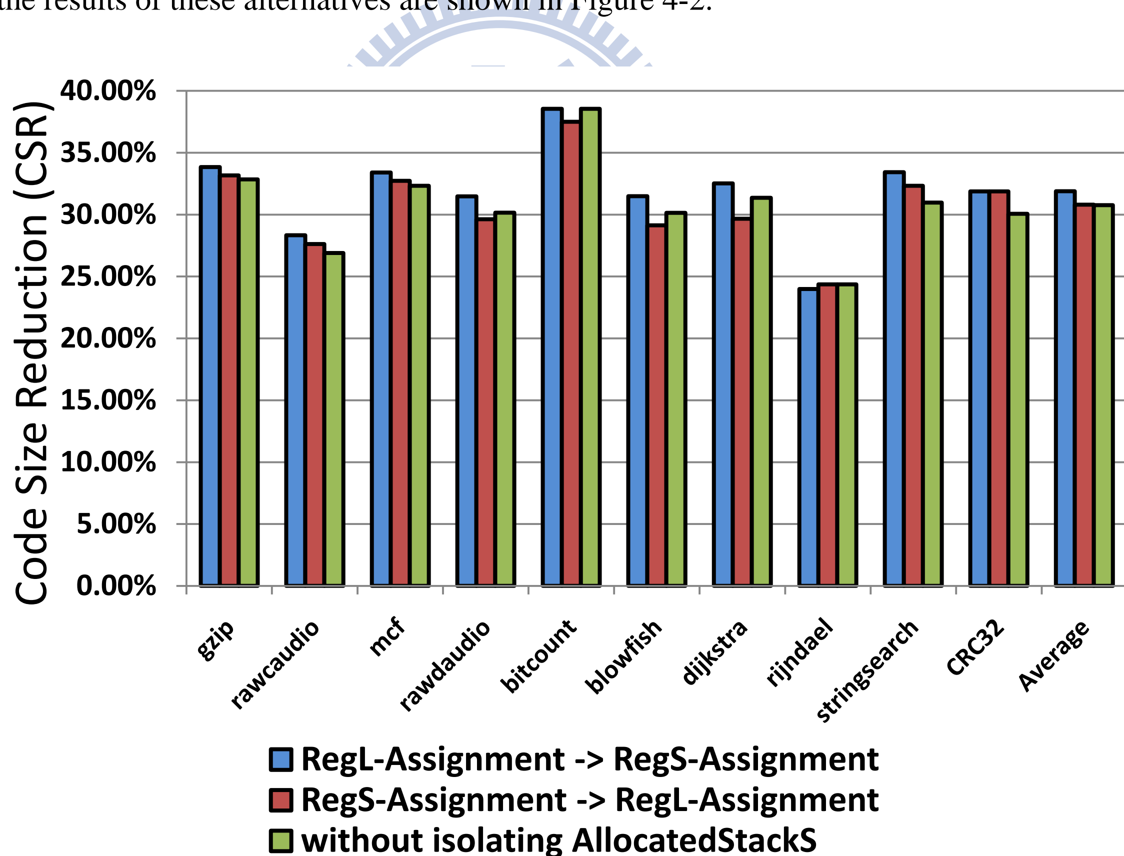


Figure 4-2 - Benchmark evaluation results of the different assignment order in MxBuGCRA.

The second experiment is the MxPrGCRA with and without "LR_L Allocation and Assignment Pass" we discussed in section 3.3.5, the result is shown in Figure 4-3. The MxPrGCRA with "LR_L Allocation and Assignment Pass" is better than without the pass in code size reduction due to the fewer spill codes generated.

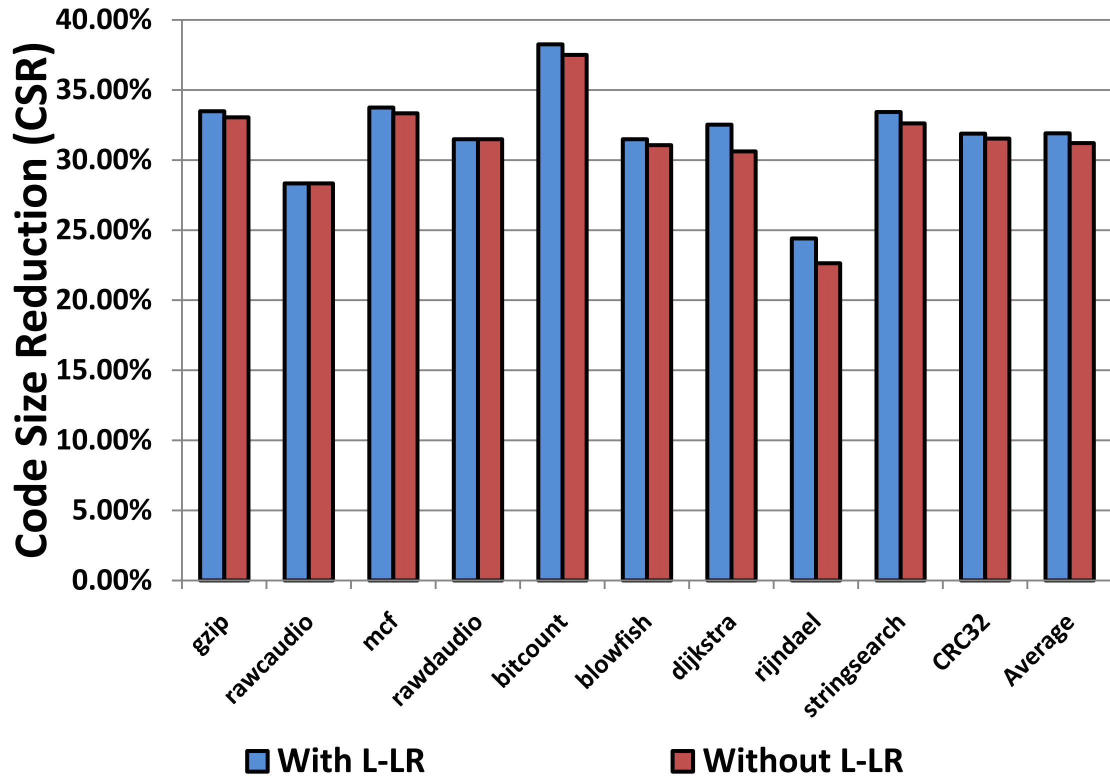


Figure 4-3 - Benchmark Evaluation Results of MxPrGCRA with/without LR_L pass.

4.2.3 Code Size Reduction

In Figure 4-4, the Y-axis indicates the code size reduction percentage, and the X-axis is the different benchmarks programs we used. The code size reduction (CSR) percentage is calculated by the following equation:

$$\mathrm{CSR}(\%) = \left(1 - \frac{\mathrm{S_InstCNT}_{\mathrm{MxBu\ (Pr\ )GCRA}} \times 16 + \mathrm{L_InstCNT}_{\mathrm{MxBu\ (Pr\ )GCRA}} \times 32}{\mathrm{Instruction\ _Count}_{\mathrm{BuGCRA}} \times 32}\right) \times 100\% \ ,$$

where S_InstCNT and L_InstCNT are the S-Format instruction count and the L-Format instruction count of the compiled program, respectively. And the base line is the program compiled by using traditional bottom-up graph coloring register allocation algorithm without Mixed-width ISA, i.e., with L-Format instructions only.

The proposed MxBuGCRA and MxPrGCRA algorithms achieve 31.89% and 31.90% of the code size reduction as shown in Figure 4-4 and Table 4-1. In other words, there are more than 60% of instructions which can be encoded in S-Format instructions. The S-Format limitation analysis will be described in Section 4.2.5. If the traditional graph coloring algorithms are used to allocate and assign registers for Mixed-width ISA, only 19.77% and 25.67% code size reduction may be obtained. Obviously, to reduce code size in mixed-width ISA processors, it is necessary to use a special heuristic model to allocate and assign registers to increase the S-Format instruction translation rate. However, since we try to spill more variables to get more INSs encoded in S-Format instructions, it possibly leads to more spill codes and may results in larger code size due to the increasing spill codes. Fortunately, the increased spill codes are very small amounts as described in Section 4.2.4.

By using our proposed algorithms, the code size reduction percentage is improved by about 12% compared to the traditional bottom-up graph coloring register allocation algorithm and about 7% compared to the traditional priority-based graph coloring algorithm. The mainly reason of the improvement is that the general algorithms allocate registers for minimizing the number of spill codes but usually assigns registers in an arbitrary way, and on the contrary, our algorithm determines a priority order to assign registers to get higher translation rate.

It is interesting that PrGCRA can get 25.67% without any consideration of Mixed-width ISA. The reason is that the priority function of PrGCRA is designed to make a variable with more cost of memory loads and stores saved has higher priority, and the number of loads and stores is similar to the access count for the variable as we mentioned in Chapter 2. Since 89% of instructions in programs are U-INS and PrGCRA assigns registers by increasing order, the $\$Var_U$s are assigned in registers which are accessible by S-Format instruction at very high probability.

Table 4-1 - Benchmark evaluation results of the proposed algorithms.

		Pronoun	CSR (%)	Spill Codes (%)
Traditional	Bottom Up GCRA	**BuGCRA**	19.77%	2.60%
	Priority Based GCRA	**PrGCRA**	25.67%	2.65%
Proposed	Bottom Up (Mix-Width) GCRA	**MxBuGCRA**	31.89%	2.84%
	Priority Based (Mix-Width) GCRA	**MxPrGCRA**	31.90%	2.88%

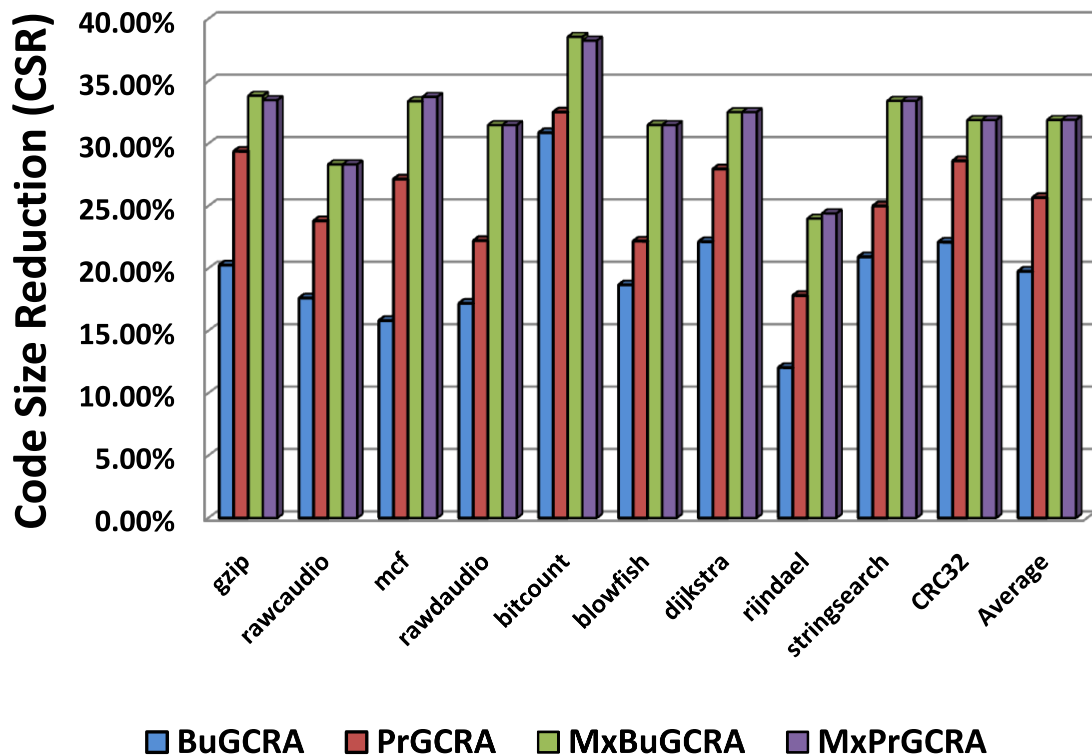


Figure 4-4 - Benchmark evaluation results of the proposed algorithms.

4.2.4 Spill Codes

Any changes in register allocation will result in different numbers of spill codes, and the number of spill codes is critical to execution performance. In this section, the number of spill codes produced by our algorithms and the traditional register allocation algorithm are evaluated and shown in Figure 4-5. And the α value is set to 0.4 for our algorithms. The Y-axis is the percentage of spill codes in programs which are compiled by different register allocation algorithms. The X-axis indicates the different benchmark programs.

The percentages of spill codes are 2.84% and 2.88% for MxBuGCRA and MxPrGCRA in average, respectively. Comparing to the traditional algorithms, the extra spill codes are 0.24% and 0.23% for MxBuGCRA and MxPrGCRA, respectively. These are really very small amounts that can be ignored.

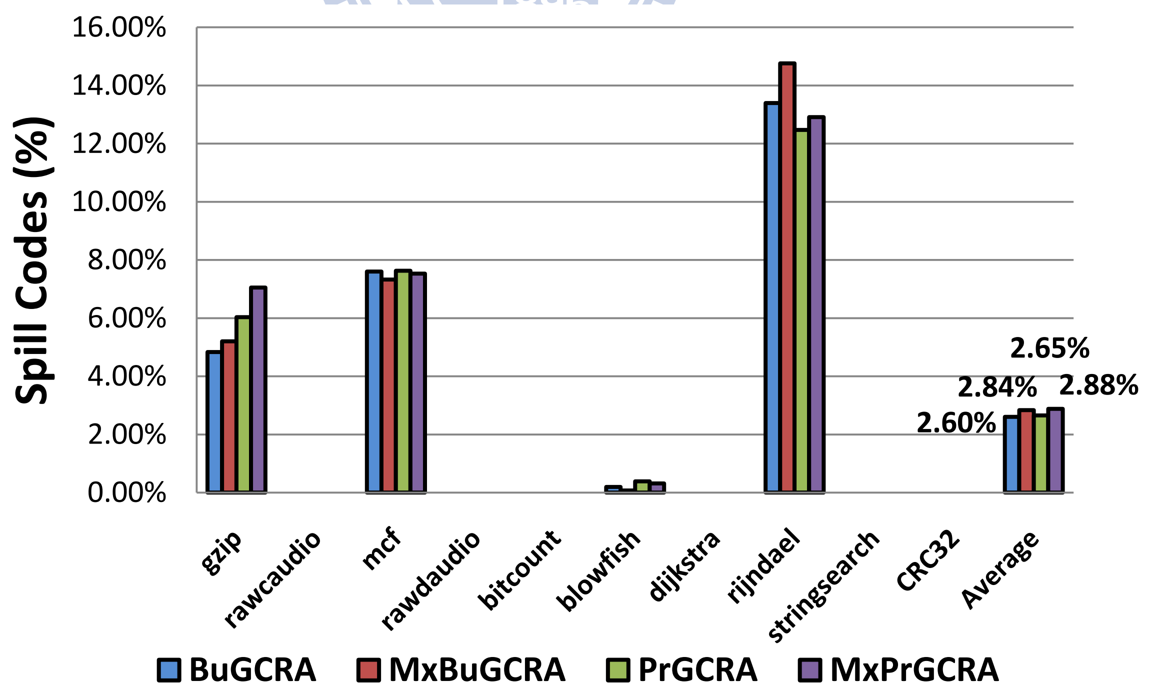


Figure 4-5 - Spill codes of the proposed and traditional algorithms.

4.2.5 S-Format Limitations Analysis

The distribution of instruction types of the traditional register allocations and our proposed algorithms has been analyzed. BuGCRA v.s. MxBuGCRA is shown in Figure 4-6, and PrGCRA v.s. MxPrGCRA is shown in Figure 4-7.

The Y-axis is the percentage of instructions in both of the two figures. In these two figures, the top blocks, green colored, are the percentage of instructions which have no operation equivalent S-Format instructions or the immediate value is oversized. The middle blocks, red colored, are the percentage of instructions each of which has one or more operation equivalent S-Format instructions but at least one of its operand registers is out of range to index of S-Format instruction. And the bottom blocks, blue colored, are those instructions whose register number and immediate value are in the range of S-Format instructions.

It is obvious that after our modification the blue sections are increased significantly. As for the red section which is about 25% in average, it is impossible to eliminate all of them because that over eight variables are alive simultaneously at many execution points in the programs, and thus, there are too many variables to assign if using only $RegisterSets_S$. By the proposed heuristic method, about 63% translation rate of S-Format instructions in entire program achieved.

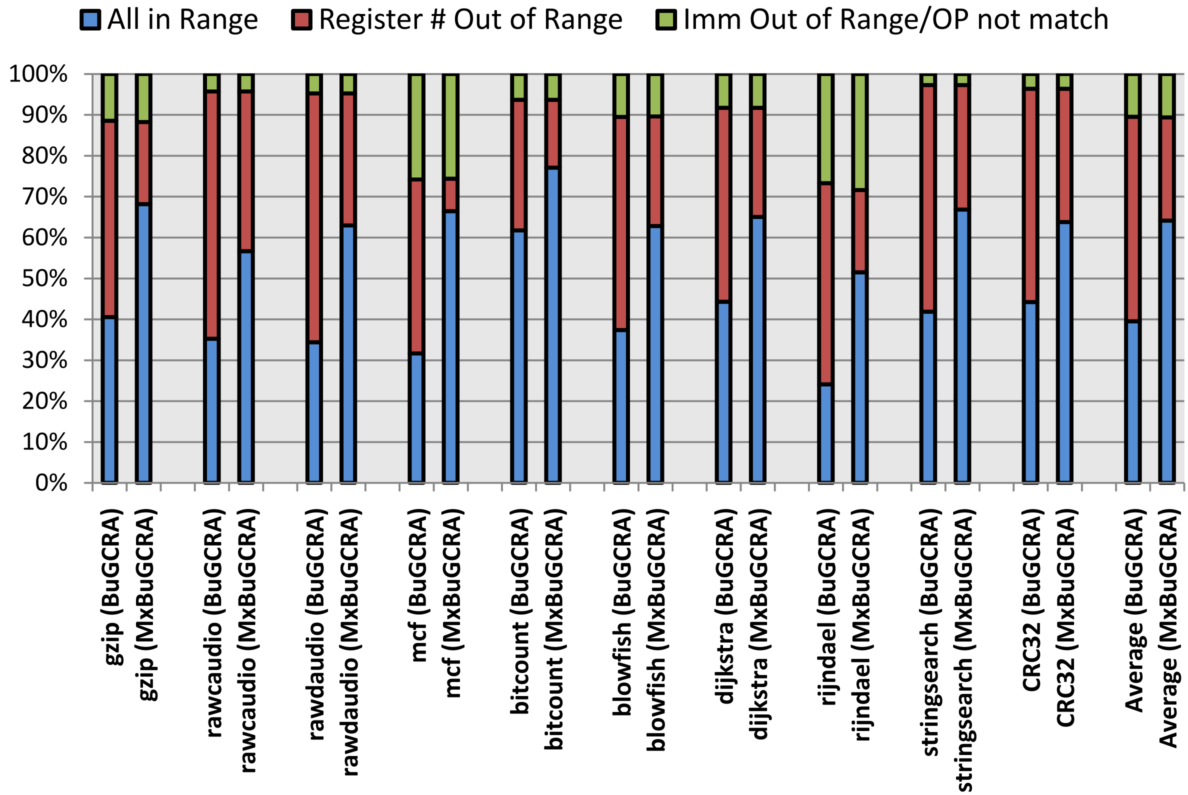


Figure 4-6 - S-Format limitation distributions - BuGCRA v.s. MxBuGCRA.

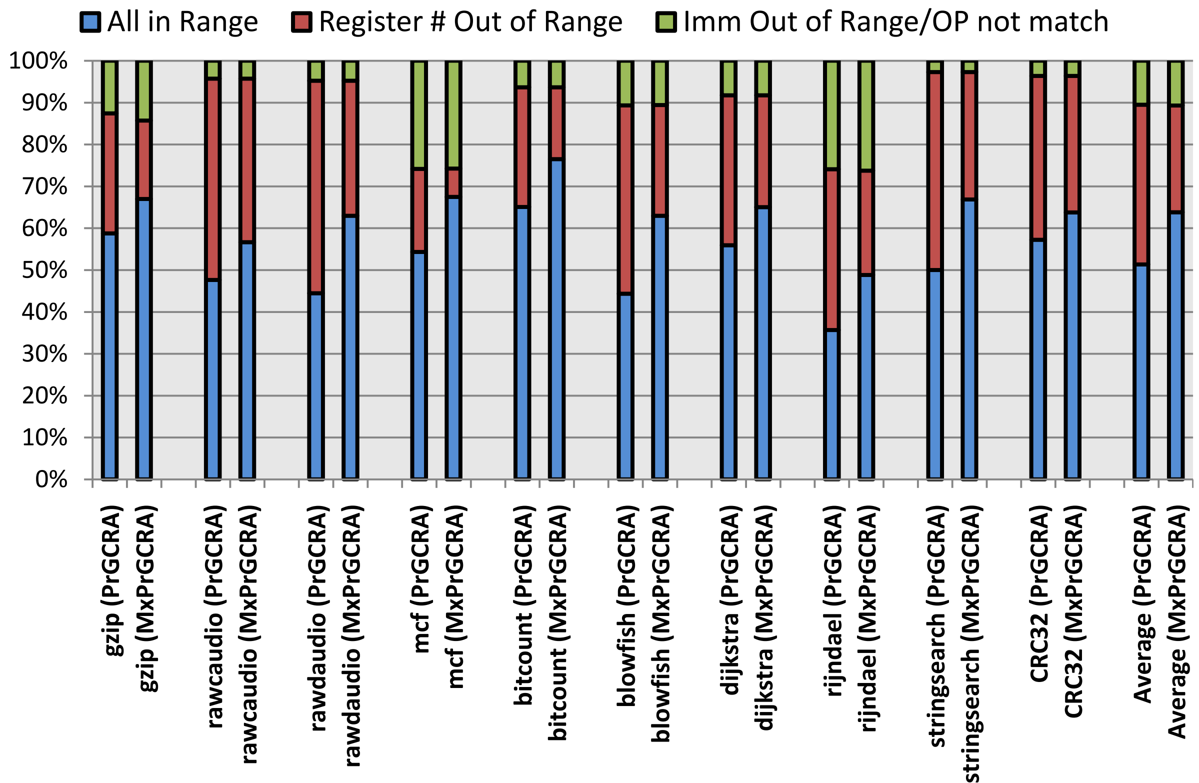
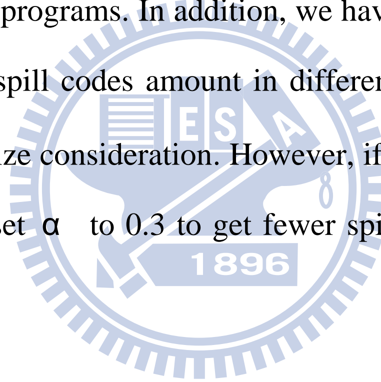


Figure 4-7 - S-Format limitation distributions - PrGCRA v.s. MxPrGCRA.

4.3 Summary for Simulation Results

In this chapter, we presented experiment results for both of the proposed algorithms and traditional algorithms. It is obvious that the approaches proposed in this thesis are easy to be modified from the traditional algorithms and effective for reducing code size by using Mixed-width ISA with mode-switch by instruction encoding.

By the statistics, the first design, MxBuGCRA, can reduce 31.89% and the second design, MxPrGCRA, can reduce 31.90% of total code size in average. As for the spill codes, the proposed algorithms with α equal to 0.4 produce spill codes in 2.84% and 2.88% in average of these benchmark programs. In addition, we have demonstrated the different results of code size reduction and spill codes amount in different α values. In this thesis, the α value is set to 0.4 for code size consideration. However, if the performance requirement is the first consideration, we can set α to 0.3 to get fewer spill codes by sacrificing a little code size reduction.



Chapter 5 Conclusions and Future Works

In this chapter, conclusions of this thesis are made first, and then the future works of this thesis are proposed.

5.1 Conclusions

As the demand for the enlarging program size due to the requirement of more program functionalities and the evolution of mixed-width ISA architectures in embedded systems, a register allocation and assignment algorithm for mixed-width ISA will become more and more important in the near future. In this thesis, we present two register allocation and assignment algorithms for mixed-width ISA with mode switch by instruction encoding.

The first proposed algorithm, MxBuGCRA, is composed of two passes: the allocation pass is to allocate registers with the consideration of minimizing spill codes, and the assignment pass is to assign register with the consideration of increasing the translation rate of S-Format instructions. The second proposed algorithm, MxPrGCRA, firstly picks variables up in an order which is considered to increase the translation rate of S-Format instructions, and then picks the remaining variables up for minimizing spill codes. Both of these two algorithms try to make efforts on reducing code size while minimizing the number of spill codes by using the sel_profit_i function for the former and the $SpillCost_i$ function for the latter.

We have conducted experiments to verify the algorithm for a mixed-width ISA processor. It is found that the code size reduction is achieved 31.89% and 31.90% on average for our algorithms respectively. Meanwhile, the proposed algorithms do efforts on minimizing the generated spill codes. From the experiments, it shows that only 0.24% and 0.23% extra spill codes generated by MxBuGCRA and MxPrGCRA than BuGCRA and PrGCRA, respectively.

The last, we also showed that the different results in code size reduction and spill codes for the proposed algorithms by setting the different α values for varies demands.

5.2 Future Works

The future works of this thesis can be put into four dimensions: reducing the runtime I-cache miss rate, obtaining the optimal solution of code size reduction, integrating other optimization techniques, e.g. instruction scheduling, for getting more code size reduction using the proposed algorithms, and modifying the proposed algorithms to match the requirements of other optimization techniques.

The first, we do not take loops into account because our objective is to reduce the program code size rather than the dynamic trace size in this thesis. However, it is a good option to observe while considering the reduction of the runtime I-cache miss rate for further researches.

The second, in order to obtain the optimal solution of code size reduction for the Mixed-width ISA with mode switch by instruction encoding, the methods which are designed for solving NP-Complete problems are good options, e.g., the Integer Linear Programming (ILP) and Partitioned Boolean Quadratic Problem (PBQP). Both of ILP and PBQP formulate problems into equations and solve them to get a unique solution. But the execution time is too slow to be adopted in practice. However, it is a good way to obtain the optimal solutions.

The third, it may have more code size reduction by integrating other optimization techniques for the proposed algorithms. For example, use a pre-instruction scheduling to shorten the live ranges of a program function. Therefore, it may release registers in $RegisterSet_S$ at some execution points and make more variables can be assigned in

$RegisterSet_S$. But any changes of instruction scheduling will affect the execution performance and register pressure. The scheduling policy should be carefully designed for these considerations.

Finally, we may get some other benefits by doing some modifications to the proposed algorithms. For example, the number of bits-change between instructions may affect the power consumption of the instruction fetch unit and the memory bus. If the Reg_S-Assignment stage of MxBuGCRA and the LR_S Allocation and Assignment pass of MxPrGCRA assign a variable into $RegisterSet_A$ when a variable in $AllocatedStacks_S$ and LR_S has no any benefits for code size reduction (i.e. no more instructions can be encoded in S-Format instructions even this variable is assigned in $RegisterSet_S$), it may reduce the bits-change between instructions.

References

[1] John Bunda, Don Fussell, W. C. Athas, and Roy Jenevein, “16-bit vs. 32-bit instructions for pipelined microprocessors”, *Proceedings of the 20th annual international symposium on Computer architecture*, San Diego, California, United States, May 16-19, 1993, p.237-246.

[2] S. Furber, *ARM System Architecture*, Addison-Wesley. 1996. ISBN 0-201-40352-8.

[3] K. Kissel, *MIPS16: High-density MIPS for the embedded market*, Tech. report, Silicon Graphics MIPS Group, 1997.

[4] Andes Technology, *Andes Instruction Set Architecture Specification*, 2008.

[5] Bor-Sung Liang, June-Yuh Wu, Jih-Yiing Lin, Ming-Chuan Huang, Chi-Shaw Lai, Yun-Yin Lien. Ching-Hua Chang, Pei-Lin Tsai, and Ching-Peng Lin, “Instruction set architecture scheme for multiple fixed-width instruction sets and conditional execution”, International Symposium on VLSI Design, Automation and Test, Sunplus Technol. Co., Ltd., Hsinchu, Taiwan, 2005.

[6] Aviral Shrivastava, Partha Biswas, Ashok Halambi, Nikil Dutt, and Alex Nicolau, “Compilation framework for code size reduction using reduced bit-width ISAs (rISAs)”, *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, January 2006, v.11 n.1, p.123-146.

[7] T. Zeitlhofer, and B. Wess, "A comparison of graph coloring heuristics for register allocation based on coalescing in interval graphs", *Proceedings of the 2004 International Symposium on Circuits and Systems*, 4: IV-529-32 Vol. 4, May 2004

[8] G. J. Chaitin, “Register allocation and spilling via graph coloring”, *Proceedings of the ACM SIGPLAN ’82 Symposium on Compiler Construction*, The Association for Computing Machinery, Boston, Massachusetts, June 1982, pages 98–105.

[9] Andrew W. Appel, *Modern Compiler Implementation in Java: Basic Techniques*,

Cambridge University Press, 1997.

[10] Fred C. Chow and John L. Hennessy, “Register allocation by priority-based coloring”, *Proceedings of the ACM SIGPLAN ’84 Symposium on Compiler Construction*, Montreal, June 1984, pages 222–232.

[11] Jonathan S. Turner, “Almost all k-colorable graphs are easy to color”, *Journal of Algorithms*, March 1988, 9(1):63–82.

[12] Chris Lattner, and Vikram Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”, *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, Palo Alto, California, March 20-24, 2004, p.75.

[13] C. Lattner et al, “The LLVM compiler infrastructure”, http://llvm.org.

[14] Preston Briggs, Keith D. Cooper, and Linda Torczon, “Improvements to graph coloring register allocation”, *ACM Transactions on Programming Languages and Systems*, May 1994, 16(3):428–455.

[15] D. Koes and S. C. Goldstein, *An analysis of graph coloring register allocation*, Technical Report CMU-CS-06-111, Carnegie Mellon University, March 2006.