國 立 交 通 大 學

資訊科學與工程研究所

碩 士 論 文

以整數線性規劃作為在混合指令集架構下之暫存器重指派方法達成程式碼減量

Code Size Reduction by Integer Linear Programming for Register Reassignment in Mixed-Width ISA Processors

研 究 生：陳柏村

指導教授：單智君 博士

中華民國 九十九 年 十 月

以整數線性規劃作為在混合指令集架構下之暫存器重指派方法達成程式碼減量

Code Size Reduction by Integer Linear Programming for Register Reassignment in Mixed-Width ISA Processors

研　究　生：陳柏村　　　　Student：Po Tsun, Chen

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

October 2010

HsinChu, Taiwan, Republic of China

以整數線性規劃作為在混合指令集架構下之暫存器重指派方法達成程式碼減量

學生：陳柏村　　　　　　　　　　　　　　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

在嵌入式系統中，程式碼大小是與效能一樣重要的議題，特別是針對記憶體受限制之系統，因此減少記憶體使用之相關研究亦是蓬勃發展。而其中一種減少記憶體使用的方式為使用混合指令集架構。這種架構通常提供二種不同長度的指令集，一般為 16 位元的短指令及 32 位元的長指令。在商業化的產品中，如 ARM、MIPS、Andes 都提出此種指令集架構，用來減少精簡指令集架構所產生之機器碼過大問題。我們的研究數據顯示，約有 49% 的指令其指令格式需至暫存器指定完後，才能決定其指令格式為 16 位元短指令或 32 位元長短令；因此如何在混合指令架構下，妥善指派暫存器是非常重要的。暫存器重指派主要是藉由重新指定暫存器編號來達成不同目的之優化；在先前的研究中，在混合指令集架構下之暫存器重指派問題已經被證明是 NP 問題，因此解決的方法大多也是用 Heuristic 的方式來得出可行解。在此篇論文研究中，我們使用了整數線性規劃來求解在混合指令架構下之暫存器重指派問題。在不考慮編譯時間的條件下，我們約可減少 34.4% 的機器碼大小。

Code Size Reduction by Integer Linear Programming for Register Reassignment in Mixed-Width ISA Processors

Student: Po-Tsun Chen   Advisor: Dr. Jyh-Jiun Shann

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

Code size issue for a memory constrained embedded system is as important as performance. There are many researches that devote to this issue. One way of reducing code size is to exploit compact instruction formats. A mixed-width ISA may provide this kind of feature for the Reduce Instruction Set Computer (RISC) processors. In general, it usually provides two different widths of instruction formats: short and long instruction format. In commercial, ARM, MIPS and Andes all support this feature in their products. From our research, the formats of 49% instructions can be decided only after register assignment. Therefore, the register assignment policy is very important for mixed-width ISA. Register reassignment renumbers the registers in each instruction for specific goals. Register reassignment for mixed-width ISA has been proved to be an NP problem. Some researches have devoted to design heuristic algorithms to find the feasible solutions. In this thesis, we use integer linear programming to solve the register reassignment problem in mixed-width ISA for reducing the code size. On the average, we reduce about 34.4% code size compared to the original program.

誌 謝

首先感謝我的指導教授 單智君老師，在這二年中不論是在報告上或是平日討論，老師對我們的細心指導及對每一個細節態度，都讓我受益良多。另外也感謝 徐慰中老師，在每一次的報告中所給與的寶貴意見。同時也感謝我的口試委員，雍忠教授、楊武教授及金仲達教授。由於教授們的指導與建議，使得這篇碩士論文更加完整及充實。

裕生學長、奕緯學長，感謝兩位學長在日常的小組討論中給與的建議及討論，特別是裕生學長，由於有你的細心指導及不厭其煩的與我討論，使得我能更順利的完成這篇碩士論文。同時也感謝實驗室的學長姐、同學們及學弟妹，因為有你們的相伴，使得研究這條艱辛的道路走起來更有趣味。

最後，感謝我的雙親及長期陪伴支持我的每一位親友，我在此獻上最大的祝福及謝意，並謹以此論文獻給各位。

陳柏村 2010/10/14

Table of Content

List of Figures

List of Tables

Chapter 1 Introduction

Embedded systems are widely used in many fields, from the hand-held devices to micro-control systems. For the cost and energy consumption consideration, embedded systems usually adopt reduced instruction set computer (RISC) processor in system. However, the features of a RISC processor make the code size larger than complex instruction set computer (CISC) processor. Therefore, many researches have devoted to reduce the code size of a RISC processor. The mixed-width ISA is one of the methods designed for this purpose. It can achieve high code density and low power consumption simultaneously. Processors, for example, ARM, MIPS, and Andes family of embedded cores, support mixed-width ISA [1-3]. MIPS proposed microMIPS as an independent 16- and 32-bit ISA that compatible with original MIPS32 and MIPS64. ARM also has thumb2 as an independent 16- and 32-bit ISA [4-5].

In general, mixed-width ISA processor is a RISC processor with more than one fixed-width instructions sets [6]. Basically, it supports two different kinds of encoding lengths for long instructions (L-Format) and short instructions (S-Format). For instance, in MIPS, 16-bit length is for the short instructions and 32-bit length for long instructions. There are some advantages of mixed-width ISAs; the most well known one is significantly reducing the code size of RISC processors. Moreover, there are some benefits corresponding to the code size reduction such as the improvement instruction cache miss rate and decreasing of the bus traffic between memory and cache. Despite of the great advantages of mixed-width ISA, some limitations of the short format instructions should be considered and we listed as follows:

1. Limited functionalities

   S-format instructions are the subset of L-format instructions and for some operations, it may use several S-format instructions to accomplish. The reason is that the operation bit field of S-format instruction cannot specify all the operations and the lengths of others fields are not enough. For example, Thumb is a subset of ARM ISA.

2. Fewer bits for holding the immediate value

   Due to ISA format limitation, an immediate value oversized the immediate field of S-format cannot be represented by S-format instruction. In MIPS16e, the length of immediate value filed is 8 bits for some S-format instructions but 16 bits for L-format instructions.

3. Fewer bits for indexing registers

   Due to ISA format limitations, the S-format instructions of some mixed-width ISAs cannot access all the registers. In MIPS, there are only 3 bits for indexing registers in 16-bit instruction but 5 bits for indexing registers in 32-bit instructions.

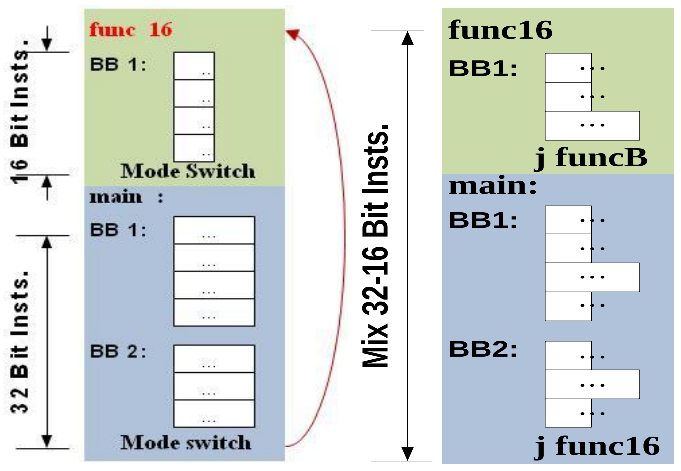
For switching long and short format instructions, there are two commonly used mechanisms as shown in figure 1-1 and we will depict in the following:

1. Using mode switch instruction:

   Compiler will insert a mode switch instruction between the L-format and S-format instructions. The mode switch instruction gives a hint to the processor which length of the instructions to be processed. To avoid frequently mode switches that cause performance loss, compiler will find contiguous convertible blocks to exploit the S-format instructions. This method is adopted in MIPS16/32 and ARM/Thumb ISA.

2. Using special instruction encoding:

   This method uses a bit in an instruction to indicate whether the instruction is L-format or S-format. Compiler or programmer can exploit each instruction which can be converted to the S-format instruction. This method is more flexible than using mode switch instruction. However, this mechanism also reduces an available bit in the ISA. In real word, microMIPS and Thumb2 all adopted this mechanism for the mixed-width ISA. We adopt this mechanism as our design in this thesis.



(a) (b)

Figure 1-1 Two different mode switch mechanisms:
(a) Use mode switch instruction (b) Use special instruction encoding

1.1 Observation

So far we have introduced the mixed-width ISA: an architecture feature for reducing code size. Figure 1.2 shows the distribution of long and short instructions if we encoded them by a MIPS like hypothetical mixed-width ISA. The experiment environment based on LLVM 2.5, and selected some benchmarks from SPEC2000, MediaBench and MiBench [7-9]. On the average, about 32.79% instructions produced by the compiler without supporting mixed-width ISA can be directly translated to the S-format instructions since these instructions are irrelative register assignments or no immediate field. There are about 19.06% instructions which cannot be translated to short instructions since the immediate value is oversized or no corresponding S-format instructions. Apart from exact long or short instructions, there are still 48.15% instructions uncertain in which formats before assigning registers. The formats of these kinds of instructions are determined by the register number. If there is a good register allocation/assignment algorithm, these instructions can be translated to S-format instructions. It still gave us a strong motivation to design better policy to handle the register assignment issue of mixed-width ISAs. If we may translate as many possibly short instructions to actual exact short instructions as possible, the code size will decrease largely.

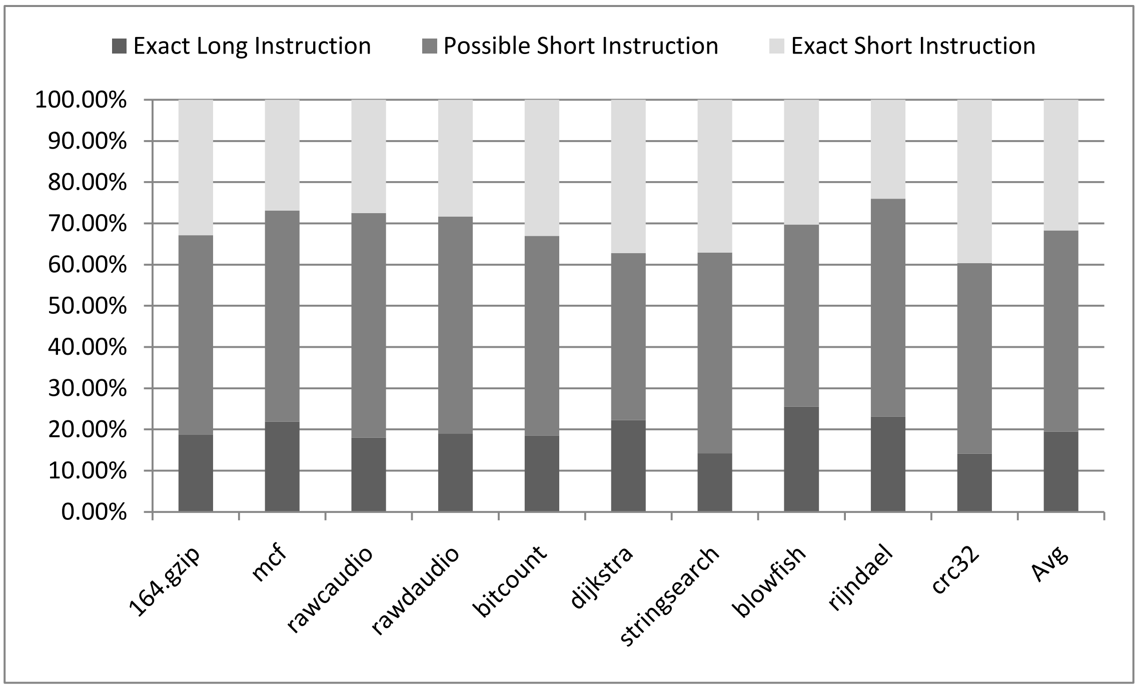


Figure 1-2 Distribution of long and short instructions

1.2 Motivation and Objective

From the previous section, it is revealed that register assignment is important for exploiting S-format instructions. Register assignment can be considered in register allocation or by inserting an optimization phase after register allocation. Previous researchers have proposed register allocation methods for mixed-width ISA processors [10-11]. However, they did not consider some important issues such as calling convention or register coalescing in their algorithms. In fact, the goal of register allocation algorithms tries to allocate variables to physical registers with minimized spilling. To make more instructions convert to S-format, it should assign more registers which can be indexed by S-format instructions. However, this two different goals may counteract by each other such that we can neither minimize the spilling and nor assign registers which can be indexed by S-format instructions adequately.

Therefore, we adopted another approach, reassigning register after register allocation. In this paper, we proposed two register reassignment methods to reduce the code size. In the first method, we reassign each used register in a function by once. For the second method, we relax the restriction of the first method by allowing the register can be reassigned more than once if the live-cycle is not overlapped.

Our objective is using the integer linear programming to formulate the register reassignment problem for mixed-width ISA processors to find the optimal translation rate of S-format instructions to reduce the code size.

1.3 Organization of this thesis

We will discuss the background knowledge and related work in Chapter 2. Two register reassignment methods by integer linear programming are introduced in Chapter 3 and 4. Chapter 5 is the experiment results and Chapter 6 is the conclusion and future work.

Chapter 2 Background and Related Work

In this chapter, we give an overview of the background knowledge for our design. Previous researches that devoted to code size reduction for mixed-width ISA are also introduced. One previous work for code size reduction by register reassignment for mixed-width ISA processors would be presented in detail and compared with ours.

2.1 Background

Some important background knowledge will be presented in following subsections including register reassignment, calling convention and integer linear programming.

2.1.1 Register Reassignment Problem

Register reassignment, also called register renumbering, can be regarded as an optimization phase after register allocation or as post-compilation. Register reassignment for mixed-width ISA has been proved to be an NP-Complete problem [12]. It reassigns registers of the given assembly code or binary code for some special purposes. The goal of register reassignment is diversity such as optimizing for code size or power consumption and so on. The unit of register reassignment can be a whole program, a function, or a basic block. Each kind of units has different advantages and is useful for different optimization purposes. In our model, the reassignment unit is a function. Since different function may have different reassignment results, it may cause the problem of the violation of calling convention.

2.1.2 Calling Convention

Calling convention is a scheme for the way of passing parameters and receiving return values between caller and callee. There are some important issues required to consider: (i) where parameters and return values are placed; (ii) the order of passing parameters; (iii) how to separate tasks between caller and callee such as setting stack pointer for function call; and (iv) which registers can be used directly without saving by callee. Different architectures may have different calling convention schemes and some target machines provide special instructions to improve the efficiency of handling calling convention. Register reassignment may lead the result of the program incorrect if not obeying the rule of the calling convention. Further discussion will be presented in Chapter 3.

2.1.3 Integer Linear Programming

Linear programming is a mathematical method to find the optimal solution under the given constraints. Linear programming has been used in a variety of areas such as computer science, management science, and so on. It formulates a problem with one objective function and one or several linear constraints. The solution of a linear programming problem must satisfy all the constraints to get the best solution. If an additional requirement that all variables of linear equations must be integer is added to linear programming, then it becomes integer linear programming (ILP). Integer linear programming has been proved to be an NP-hard problem. The canonical form of the integer linear programming is listed as follows:

$$\begin{cases} \text{minimize } f(x_1, x_2, \dots, x_n) = c_1x_1 + c_2x_2 + \cdots + c_nx_n + d, \\ \text{subject to } \sum_{i=1}^{n} a_{ji}x_i \le b \text{ for } j = 1, \dots, m \\ x_i \ge 0 \text{ for } i = 1, \dots, n \ (x_i \text{ is an integer}) \end{cases}$$

Another variation of integer linear programming is binary integer linear programming (BILP). BILP is an integer programming problem that contains only binary variables and is also called 0-1 integer linear programming. In this thesis, we will use BILP to model our register reassignment problems.

2.2 Related Work

Most of the previous researches to reduce code size for mixed-width ISA with mode switch instructions have proposed different kinds of compilation methods [13-16]. These mechanisms require finding contiguous convertible block for translating to short instructions and insert mode switch instructions between those blocks. It may cause the two following situations: the code size reduction rate cannot be deeply exploited and the mode switch instruction may lead performance losses. Our approach adopts the mixed-width ISA with special instruction encoding that allows the freeform of mixing short and long format instructions. Hence, the above problems are not needed to address in our method.

For the mixed-width ISA with special instruction encoding, Tobias J. K. Edler von Koch et al. proposed a feedback-guided approach that cooperated instruction selection and register allocation methodology to exploit the short instructions of original program [17]. Their method based on three steps; the first step is IR annotation phase that maps mid-level intermediate representation (MIR) to low-level intermediate representation (LIR) by giving preference to short format instructions and annotates extra spill or move code to mid-level intermediate representation. The next step maps virtual registers to physical registers and then decides the LIRs inserted for reducing register pressure. The third step is feedback-guided code generation that deactivates the LIRs produced for reducing register pressure and reproduces the LIRs with long format preference. Subsequently register allocation is performed, and finally, native code interleaving with long and short format instructions are emitted. Although this is a very creative method and improved the performance, this method still suffers from high register pressure such that the code size reduction rate is not ideal.

Our method reduces code size by register reassignment. Register reassignment for mixed-width ISA has been proved as an NP-Complete problem [12]. One previous research of code size reduction by register reassignment has proposed heuristic methods to solve the problem [18]. In our method, we put efforts on using ILP to model the register reassignment problem of mixed-width ISA to find the optimal solution of code size reduction under the given constraints. We will present this related work in the next subsection.

2.2.1 Ku's Method

In Ku's heuristic methods [18], there are three main phases:

1. Register selection

2. Mapping and reassignment assessment

3. Instruction insertion

For the register selection phase, they proposed two register selection methods which interchange the selected registers with small set registers. Note that the small set register is a set of registers which S-format instructions can be indexed. Those methods are:

1. Selection registers by the priority that based on registers' usage counts
2. Selection registers by the priority that based on registers' usage counts and neighbor graph.

The first method is very simple. It counts the number of times for each register used in the input function. The usage count of each register is regarded as the priority of the register for mapping into the small set registers which S-format instruction can be indexed.

Obviously, the first method is too trivial. To increase the opportunity of the used registers appeared in the same instruction, the second method build a new data structure called neighbor graph for manipulating the priority of the candidate registers. The neighbor graph is a special data structure with vertex, edge, weight and priority.

A vertex of a neighbor graph stands for a register number used in the input function and an edge between two vertices represents that these two registers have appeared in same instructions. The weight on an edge is the appearance counts of the two registers. The priority of a vertex is initialized by the usage count of a register. Once a register is selected to map into small register set, the priorities of other vertices which connected with the selected register will be updated by adding the weight of the edge of the selected register to increase the opportunity of related registers. Figure 2-2 is an example of the neighbor graph. Assume that there are two instructions and four used registers, \$2, \$3, \$14, and \$16 are used. The initial priorities of these four registers are 12, 6, 4, and 3. For the first instruction, we can find \$2, \$3, and \$14 appeared. Therefore, these three registers form three vertices and there are three edges connected with each other. Moreover, the weight of these three edges is 1. For the next instruction, \$2, \$3, and \$16 appear in the same instruction. Since \$16 didn’t appear in previous instruction, we add a new vertex for it and insert two edges between \$2, \$16 and \$3, \$16. \$2 and \$3 have appeared in previous instruction, thus, we only need to increment the weight of edge between \$2 and \$3 by one.

The next phase is mapping and reassignment assessment. In this phase, the selected registers will be mapped to small register set. The mapping rules are: if the selected registers belong into small set registers, it would be mapped to itself without changing. Otherwise, these registers are mapped from highest to lowest priority through interchanging selected registers with the rest of small set registers. After mapping completion, it started out analyzing the mapping result of a function to determine whether it is valuable to reserve. In consideration of calling convention, it may need to insert some code before and after each function call. Therefore, it may make the code size increased. In Ku’s method, the mapping result is accepted only if the code size reduction was larger than the extra overheads.

The final phase is designed for dealing with the calling convention problem. As we know, register reassignment may violate the calling convention, so it has to insert some code to keep the correctness of the program. For the different categories of registers such as caller-saved or callee-saved, each category has different code insertion rules.

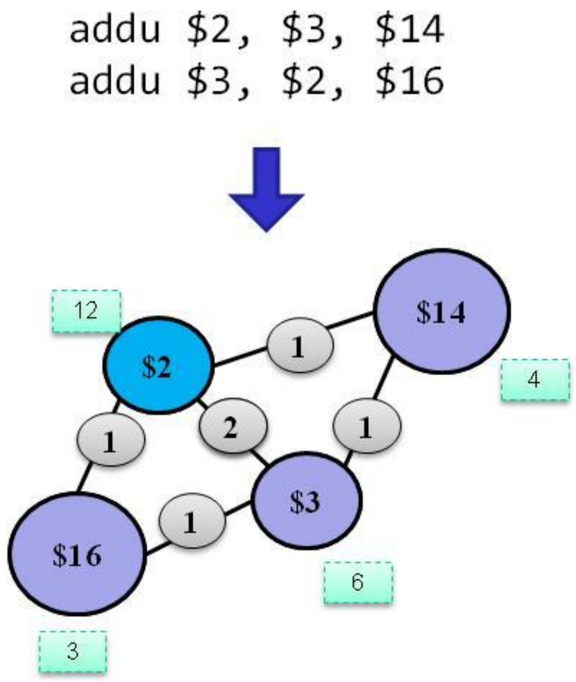


Figure 2-2 Example of neighbor graph

2.2.2 Discussion of Ku’s Method

Ku’s method is very simple and efficient. However, for our purpose, it may be too rough to solve the problem. We discuss the drawbacks of his method in this subsection.

The first drawback of Ku’s method is that it didn’t deeply consider the reassignment overheads. Although the registers are selected according the usage count and neighbor-graph, , his method did not consider the calling convention overheads while selecting registers.

Another important issue is that it interchanged the registers of mapping pairs. This is an easy way to deal with the selected registers that map to the small set registers. However, as we have mentioned in previously, different reassignment combinations may have different reassignment costs under the calling convention. Therefore, interchanging the mapping pair may not cover all possible cases. The other is that this method does not analyze the liveness of the registers such that the calling convention overheads cannot be computed accurately. Brief comparisons between Ku's method and our method are listed in table 2-1.

Table 2-1 Proposed design v.s. Ku's method

	Ku's Method	Our method
Method	Heuristic	Optimal
Time Complexity	Low	High
Reassignment restriction	Strict	No Restrict
Reassignment Overhead Estimation	Conservative	Accurate

Chapter 3 ILP Formulation for Register Reassignment Method Ⅰ

In this chapter, we will present how to model register reassignment problem as an integer linear programming. Firstly, an overview of the proposed design and the assumptions of our model are given. After the notations are defined, we will discuss all the details of our model.

3.1 Overview

Figure 3-1 shows the proposed design of our register reassignment method. After compiler front-end processing, it produces intermediate representation, (IR) and then, enters into back-end processing. The first stage of back-end processing is instruction selection, which maps IR to target instructions. The second stage is instruction scheduling which takes responsibility for arranging instructions order to reducing stalling or hazard. Then register allocation maps virtual registers to architecture registers. We add a register reassignment phase after register allocation. In this phase, we analyze each instruction of a function and construct the control flow graph in order to computing the liveness information of the physical registers. Besides the liveness information, we compute the reassignment overheads here. Then, we apply our ILP model for register reassignment is applied to produce the constraints and objective function. After an integer program is produced, a third party ILP solver library is used to solve the problem. Finally, we retrieve the results of register reassignment from the solver and the instruction formatting is called to determine the correct format for producing the assembly code. We will present the details of the ILP model in the following sections.

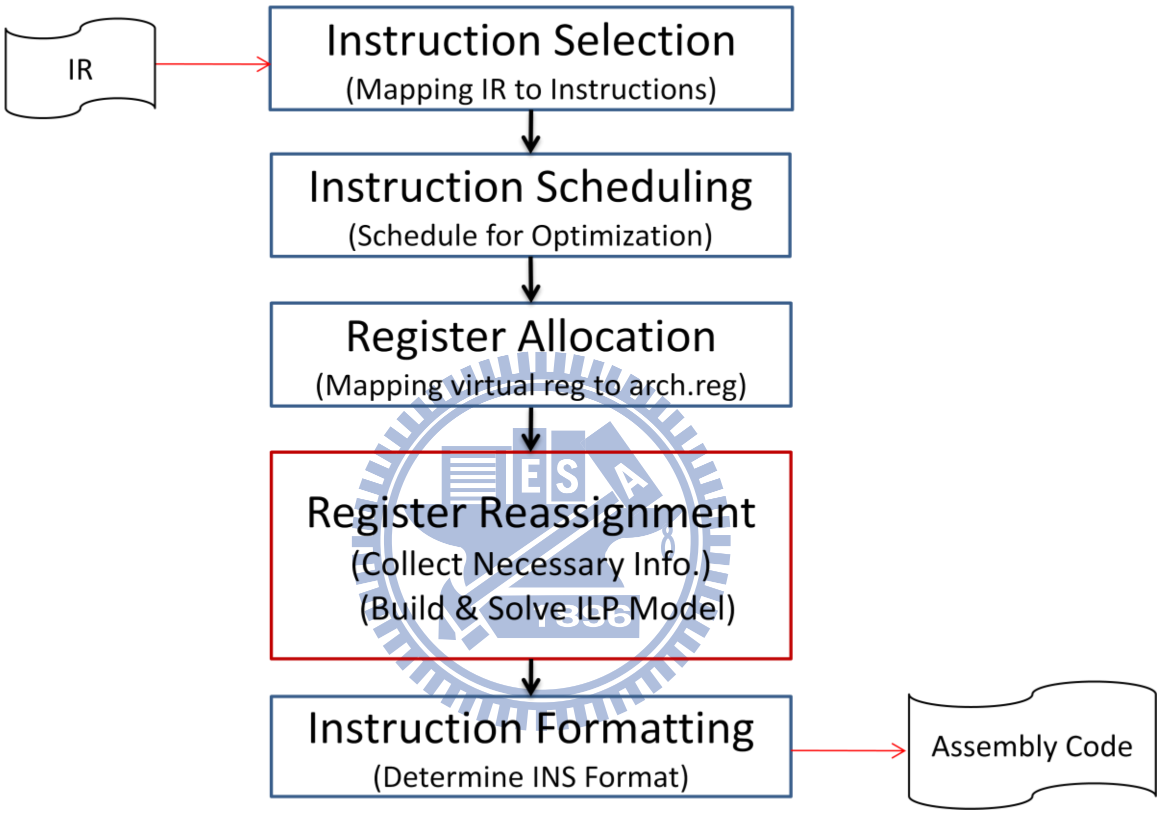


Figure 3-1 Proposed flow chart of our register reassignment method

3.2 Description of Register Reassignment

In this subsection, we will introduce the assumptions, input and output of our ILP method 1. We assume that our target architecture has only two kinds of instruction widths, one long instruction and one short instruction formats. Input of our model is the instructions of a function. The basic register reassignment unit of our method is a function. All the instructions of a function, except those with certain long or short format by arbitrary register assignment, are analyzed. To simplify the problem, we also make an assumption in our first method that each register can only be reassigned to only one register after reassignment. After solving the problem, the produced reassigned register numbers will be used to rewrite the corresponding instructions.

3.3 Notations

The necessary definitions used in our model will be introduced in this section and are classified into ISA features, function features and reassignment matrix.

3.3.1 ISA Features

ISA features related to our model are (i) the target register file, (ii) ISA format, (iii) calling convention, and (iv) classifications of instructions.

Two sets and one function defined for target register file are described as follows:

- $RegisterSet_A$: set of all the physical registers except special purpose registers in the target architecture.
- $RegisterSet_S$: the set of all the physical registers which can be indexed by S-format instructions.
- isSmallReg(Reg), Reg $\in$ $RegisterSet_A$: a boolean function which represents whether the given register will be reassigned to $RegisterSet_S$.

The following variables are related to ISA format.

- $\text{InstSize}_{\text{long}}$: the number of bytes of a long instruction.

- InstSize$_{Short}$: the number of bytes of a short instruction.
- SavedSize: the number of saved bytes when a long instruction is translated to a short instruction.

The following sets are related to the calling convention.

- *CalleeSavedRegs*: the set of the physical registers saved by callee.
- *CallerSavedRegs*: the set of the physical registers saved by caller.
- *TempRegs*: the set of physical registers which are used for storing temporary value.
- *ArgRegs*: the set of physical registers which are used for passing arguments.
- *RetRegs*: the set of physical registers which are used for passing return values.

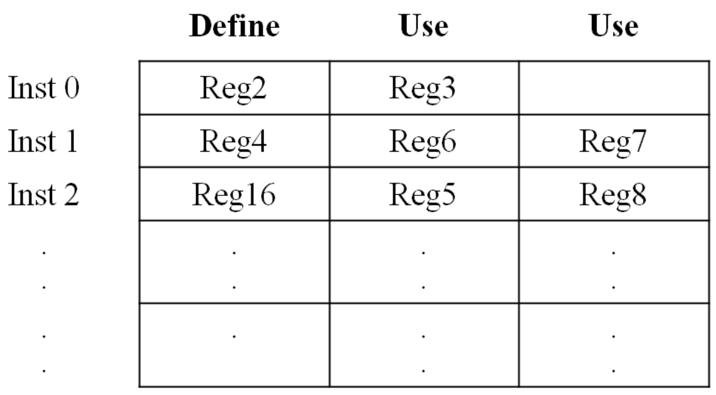
The following sets defined for the classification of input instructions.

- *L_INS*: the set of certain L-format instructions. An instruction which has no equivalent short format instruction or has oversized immediate value for the equivalent short format instruction belongs to this category.
- *S_INS*: the set of certain S-format instructions. An instruction which has an equivalent short format instruction by arbitrary register assignment belongs to this set.
- *U_INS*: the set of instructions each of which may be either L-format or S-format according to the register assignment. This kind of instructions is the input of our register reassignment method.

3.3.2 Function Features

In this subsection, we define the variables or sets to represent function features. Those are (i) register usage of a function and (ii) register number used by each instruction of the function.

- *RegUsed*: the set of registers used in the function.
- *RegNotUsed*: set of registers not used in the function.
- RegOpndCount(Inst), Inst $\in$ *U_INS*: a function which returns the number of register operands of an uncertain format instruction Inst.
- Inst, Inst$\in$*U_INS*: instruction variable of each *U_INS*. Inst = 1 when Inst can be converted to S-format instruction after reassignment.
- InstTable(Inst, k), Inst $\in$ *U_INS*; $1\leq k \leq$ RegOpndCount(Inst): a function that returns the kth register number used by Inst. Figure 3.2 shows the structure of InstTable. For example, if we use InstTable(Inst1, 1), this function will return Reg4 as the output.



	Define	**Use**	**Use**
Inst 0	Reg2	Reg3	
Inst 1	Reg4	Reg6	Reg7
Inst 2	Reg16	Reg5	Reg8
. .	. .	. .	. .
. .	.	. .	. .

Figure 3-2 Structure of InstTable

3.3.3 Reassignment Matrix

To easily define the constraints for our model, we first introduce a special matrix called reassignment matrix RM. Figure 3.3 shows the structure of an RM. For each row of RM, R_{bi}, $1 \le i \le m$, stands for the register number before register reassignment; for each column of RM, $R_{aj,}$ $1 \le j \le n$, stands for the register number after register reassignment. Each element in RM is denoted as $X_{i,j}$. The $X_{i,j}$ is define as following:

X_{i_j}: i, j $\in$ $RegisterSet_A$: each element of RM, $X_{i\_j} = 1$ if register i reassigned to register j

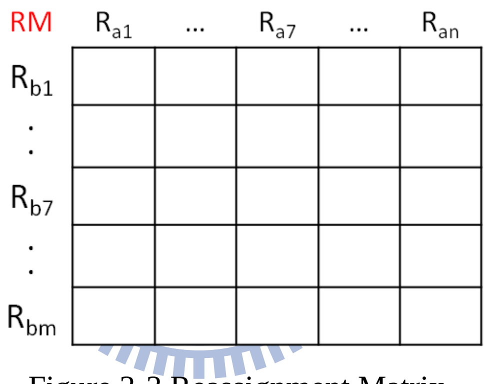


Figure 3-3 Reassignment Matrix

3.4 Formulation

After defining the necessary notations used in our ILP model, we present the model for register reassignment problem in this section. We first show the objective function and then introduce the constraints of register reassignment.

3.4.1 Objective Function

If all the register operands of an L-format instruction belong to small set registers, it can be represented by S-format instruction and thus the code size is reduced. However, reassigning registers may also make program incorrect. Since the basic reassignment unit of our model is a function, it means that different functions may have different reassignment results. In other words, the results may violate the calling convention such that the program is incorrect. Therefore, we may need to insert extra code to make our program correct. However, code insertion counteracts the benefits of the register reassignment, i.e., it is a trade-off between the both. Hence, the objective function of our model is expressed as shown in Eq. (1):

$$\textbf{Maximize} \qquad \mathrm{CSR} \qquad (1)$$

$$\text{where } \mathrm{CSR} = \mathrm{CR_{short}} - \mathrm{Reassignment_{overheads}}$$

$\mathrm{CR_{short}}$ represents code size reduction and $\mathrm{Reassignment_{overheads}}$ shows the extra code insertion for the calling convention.

A. $\mathrm{CR_{short}}$

The formulation of $\mathrm{CR_{short}}$ is listed as following:

$$\mathrm{CR_{short}} = \sum_{\mathrm{Inst} \in U\_INS} \left(\prod_{\mathrm{k=1}}^{\mathrm{RegOpndCount(Inst)}} \mathrm{isSmallReg(InstTable(Inst, k))} \right) \times \mathrm{SavedSize}$$

The above is a nonlinear equation when the register operands of an instruction are larger and equal than two. For this kind of nonlinear boolean function, we can use linearization method to deal with it through replacing the nonlinear part of the function by new variables and adding new constraints to linearize this objective function [19]. The modified objective function after linearization is listed in the following.

$$\mathrm{CR_{short}} = \sum_{\mathrm{Inst} \in U\_INS} \mathrm{Inst} \times \mathrm{SavedSize}$$

The added constraints are listed as follows:

$$\mathrm{Inst} - \mathrm{isSmallReg}\big(\mathrm{InstTable}(\mathrm{Inst}, \mathrm{k})\big) \leq 0$$

$$\sum_{\mathrm{k}=1}^{\mathrm{RegOpndCount(Inst)}} \mathrm{isSmallReg}\big(\mathrm{InstTable}(\mathrm{Inst}, \mathrm{k})\big) - \mathrm{Inst} \leq \mathrm{RegOpndCount}(\mathrm{Inst}) - 1$$

B. Reassignment Overheads

Register reassignment may violate the rule of calling convention since each function may have different reassignment results. Therefore, we may need to insert extra code to keep the correctness of the program. In our model, possible register reassignment overheads of each function can be computed in advance. To deal with the calling convention, we classify the general purpose registers into two categories according to whether the registers which will be preserved and not preserved across function call. These two categories are callee-saved registers and caller-saved registers. In addition, caller-saved registers can further be classified into temporary registers, argument registers and return value registers. Different solution of register reassignment would introduce different overheads. To explain the case of reassignment overheads analysis, a conceptual basic function layout after compiler code generation like Figure 3-4 is used. After compiler completing register allocation, the virtual registers will be mapped to the physical registers. Compiler will insert prolog code and epilog code to preserve and restore the callee-saved register used in current function, respectively.

Moreover, if the content of caller-saved registers would be used after a function call in current function, compiler will insert preserved code before the function call and retrieved code after function call. In a RISC processor, the calling convention define the argument registers or return value registers to pass parameters or receive the return values. For analyzing the reassignment overheads, those are important and affect the reassignment overheads significantly.

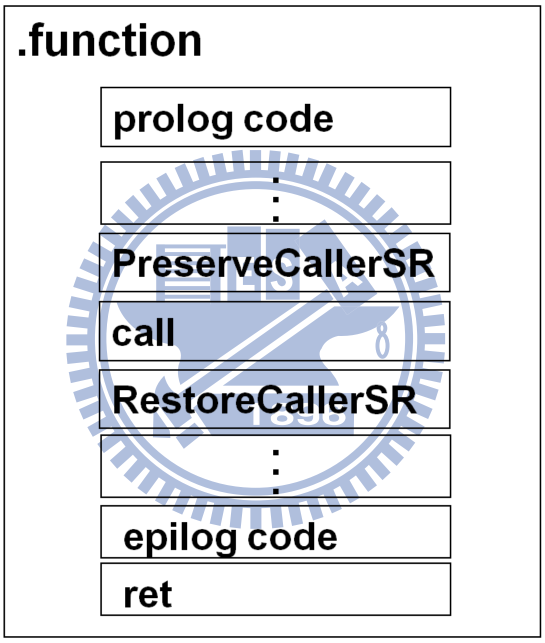


Figure 3-4 Function layouts after code generation

To compute reassignment overheads, we need to know the type of registers before and after reassignment, the function prototype and the liveness of each used register. In assembly code, function prototype information is very hard to retrieve or analyze. In this thesis, we implement our design in the LLVM compiler [22] to retrieve the necessary information such as function prototype. The other important information we need to know is live range of each used physical registers. To achieve this goal, we construct the control flow graph to compute the live-in, live-out, define and use information of each instruction of the function to calculate to calculate liveness of each used register.

The variables used to check the register type before and after reassignment are defined as follows:

- isCalleeToCallee(R_{bi},R_{aj}), R_{bi}, $R_{aj} \in$ RegisterSet$_A$: a boolean function which returns whether a callee-saved register is reassigned to a callee-saved register.
- isCalleeToCaller(R_{bi},R_{aj}), R_{bi}, $R_{aj} \in$ *RegisterSet$_A$*: a boolean function returns whether a callee-saved register is reassigned to a caller-saved register.
- isTempToCallee(R_{bi},R_{aj}), R_{bi}, $R_{aj} \in$ *RegisterSet$_A$*: a boolean function returns whether a temporary register is reassigned to a callee-saved register.
- isTempToCaller(R_{bi},R_{aj}), R_{bi}, $R_{aj} \in$ *RegisterSet$_A$*: a boolean function returns whether a temporary register is reassigned to a caller-saved register.
- isArgToCallee(R_{bi},R_{aj}), R_{bi}, $R_{aj} \in$ *RegisterSet$_A$*: a boolean function returns whether an argument register is reassigned to a callee-saved register.
- isArgToCaller(R_{bi},R_{aj}), R_{bi}, $R_{aj} \in$ *RegisterSet$_A$*: a boolean function returns whether an argument register is reassigned to a caller-saved register.
- isRetToCallee(R_{bi},R_{aj}), R_{bi}, $R_{aj} \in$ *RegisterSet$_A$*: a boolean function returns whether a return value register is reassigned to a callee-saved register.

- isRetToCaller(R_{bi},R_{aj}), R_{bi}, $R_{aj} \in$ *RegisterSet$_A$*: a boolean function returns whether a return value register is reassigned to a caller-saved register.

The following definitions are the overheads for the reassignment of different types of registers:

- CostofCalleetoCaller: overheads of reassignment callee-saved to caller-saved registers.
- CostofCalleetoCallee: overheads of reassignment callee-saved to callee-saved registers.
- CostofTemptoCallee: overheads of reassignment temporary to callee-saved registers.
- CostofTemptoCaller: overheads of reassignment temporary to caller-saved registers.
- CostofArgtoCallee: overheads of reassignment argument to callee-saved registers.
- CostofArgtoCaller: overheads of reassignment argument to callee-saved registers.
- CostofRettoCallee: overheads of reassignment return value to callee-saved registers.
- CostofRettoCaller: overheads of reassignment return value to caller-saved registers.

Now we start out presenting the reassignment overheads computation. We will discuss each reassignment case by case.

Case 1: Reassignment of a callee-saved register.

For reassigning a callee-saved register to a callee-saved registers, we may modify the stack offset only since the callee-saved register has been preserved at the prolog and restored at epilog, respectively. The modification of code is shown in Figure 3-5.

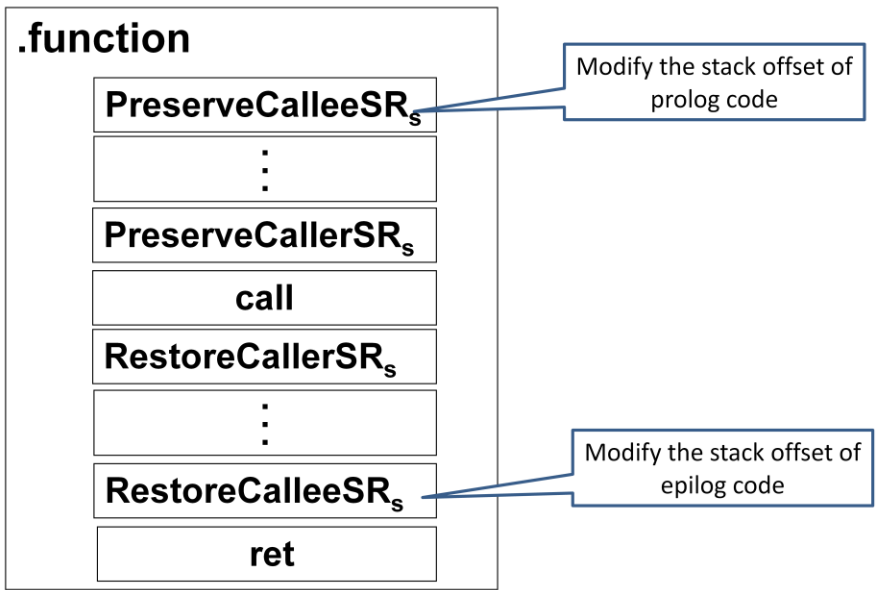


Figure 3-5 An example of reassignment a callee-saved register to a callee-saved register

For reassigning a callee-saved register to a caller-saved register, we need to remove the code for preserving and restoring callee-saved register at prologue and epilogue. Also, for each function call in the current function, if the live ranges of original callee-saved registers cross the function call, the preserved and restored code of the register before and after the function call should be inserted. The modification of code is shown in Figure 3-6.

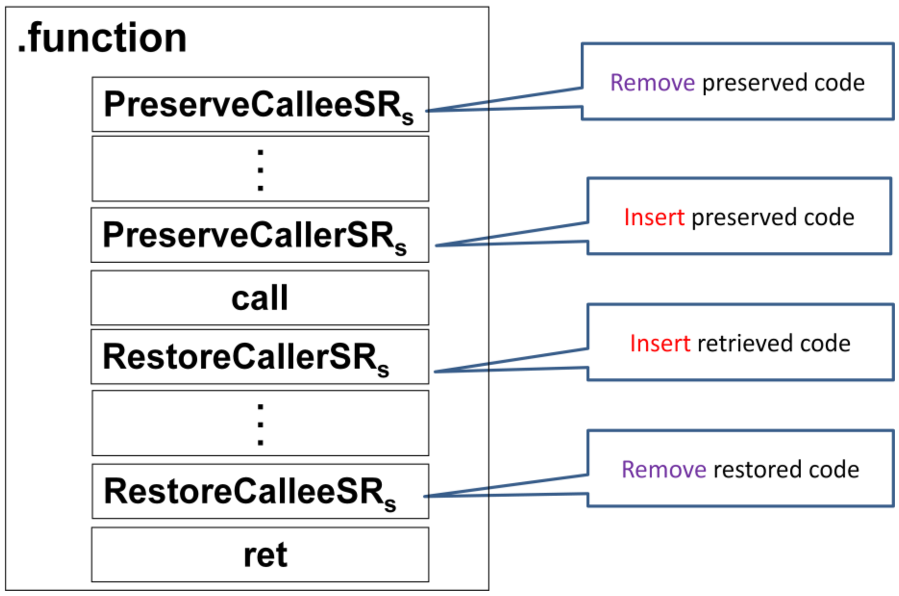


Figure 3-6 An example of reassignment a callee-saved register to a caller-saved register

Case 2: reassignment of a temporary register.

For reassigning a temporary register to a callee-saved register, it needs to insert code at prologue and epilogue for preserving a callee-saved register. If a temporary register spill/reload the value before/after each function call, we need to remove the code. Figure 3-7 shows an example of a temporary register reassigned to callee-saved registers.

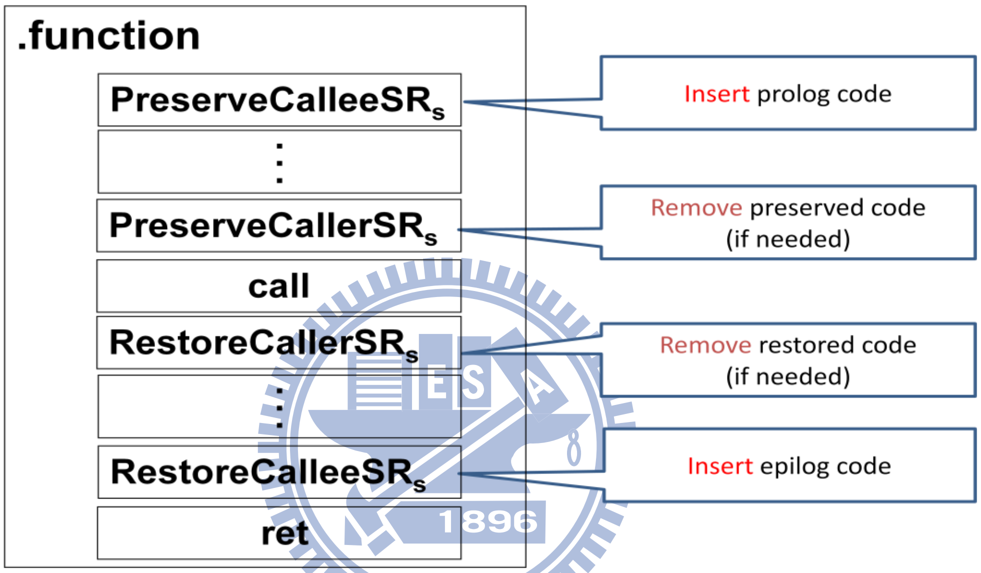

.function
$\text{PreserveCalleeSR}_s$
⋮
$\text{PreserveCallerSR}_s$
call
RestoreCallerSR_s
⋮
RestoreCalleeSR_s
ret
Insert prolog code
Remove preserved code (if needed)
Remove restored code (if needed)
Insert epilog code

Figure 3-7 An example of reassignment a temporary register to a callee-saved register

For reassigning a temporary register to a caller-saved register, if a temporary register has been preserved and retrieved before/after each function call, only the stack offset need to be modified and no insertion of code is required. Figure 3-8 shows the example of reassignment temporary registers to caller-saved registers.

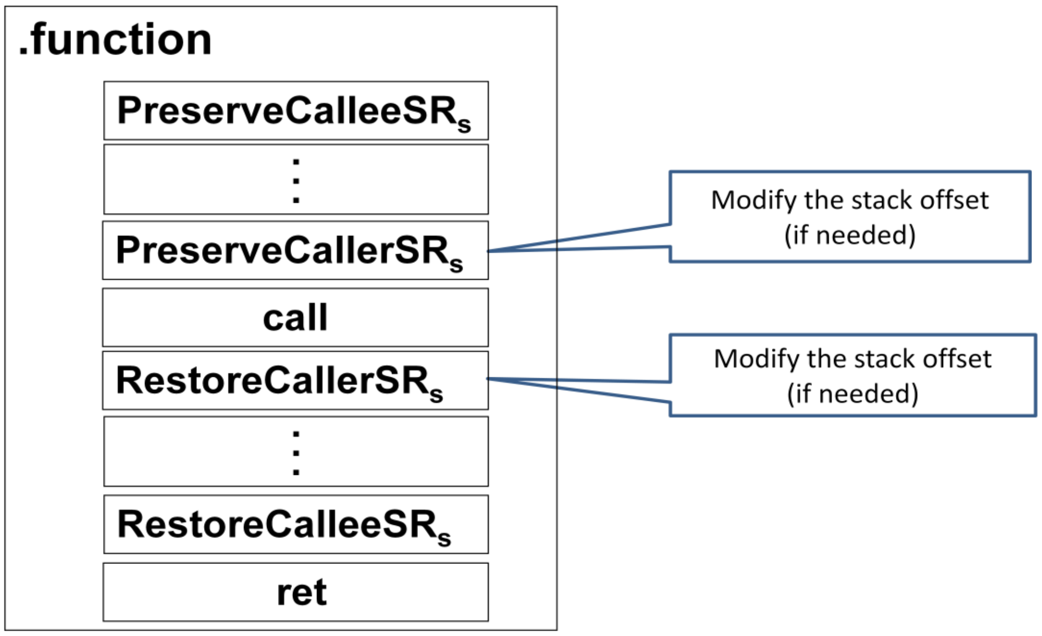


Figure 3-8 An example of reassignment a temporary register to a caller-saved register

Case 3: reassignment of an argument register

For reassigning an argument register to a callee-saved register, we need to insert code at prologue and epilogue for preserving and restoring the content of the callee-saved register and we also need to insert an instruction to move the argument of the current function to the reassigned register. If the argument register has been preserved and retrieved before and after each function call, it should be removed. For each function call, it also needs to check whether the argument register has been used for passing parameter to the called site. If it is used, a move instruction has to be inserted to move the value to the argument register. Figure 3-9 shows an example of reassignment an argument register to a callee-saved register.

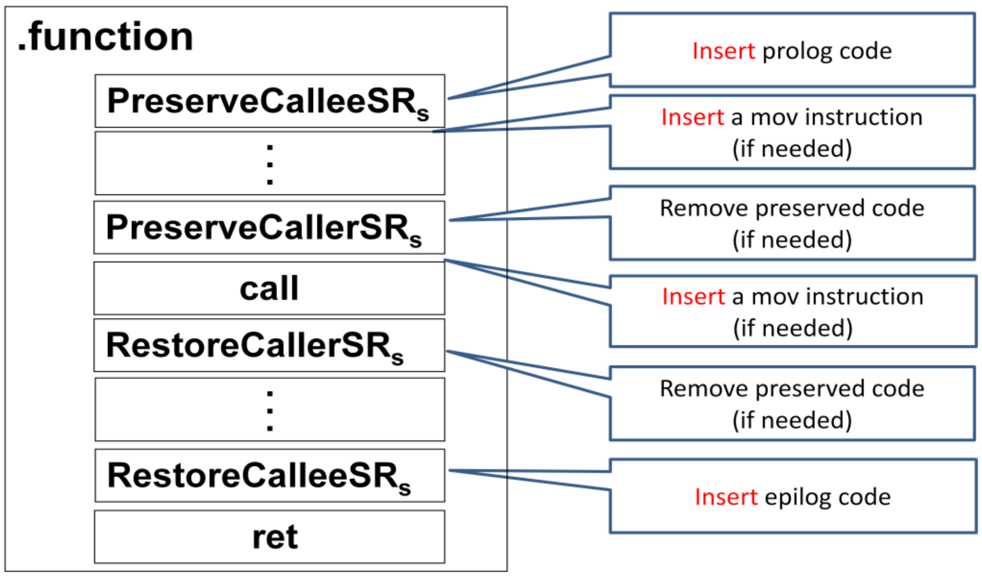

.function
$PreserveCalleeSR_s$
⋮
$PreserveCallerSR_s$
call
$RestoreCallerSR_s$
⋮
$RestoreCalleeSR_s$
ret
Insert prolog code
Insert a mov instruction (if needed)
Remove preserved code (if needed)
Insert a mov instruction (if needed)
Remove preserved code (if needed)
Insert epilog code

Figure 3-9 An example of reassignment an argument register to a callee-saved register

For reassigning an argument register to a caller-saved register, we have to insert move instructions for an argument passed into the current function and parameter passed to its function call. If the caller-saved register spills/reloads the value before/after function call, the stack offset should modified. Figure 3-10 shows an example of the reassignment of an argument register to a caller-saved register.

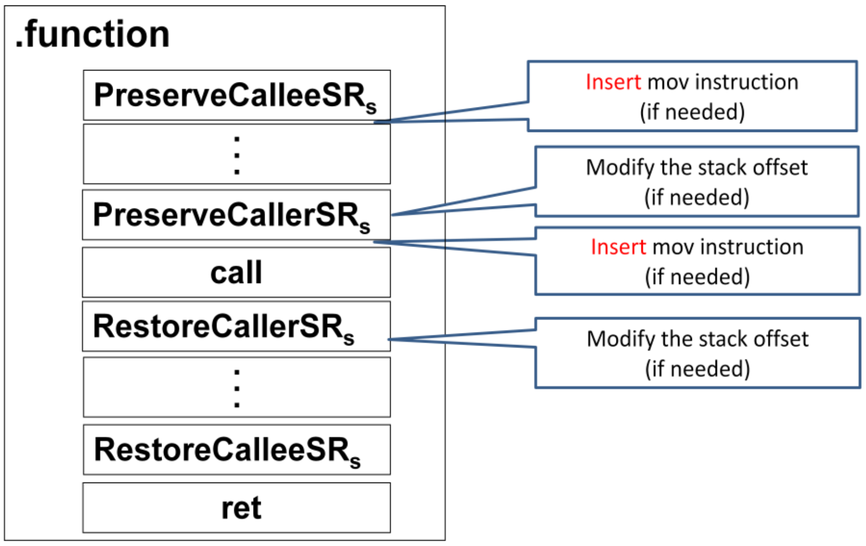

.function
$PreserveCalleeSR_s$
⋮
$PreserveCallerSR_s$
call
$RestoreCallerSR_s$
⋮
$RestoreCalleeSR_s$
ret
Insert mov instruction (if needed)
Modify the stack offset (if needed)
Insert mov instruction (if needed)
Modify the stack offset (if needed)

Figure 3-10 An example of reassignment an argument register to a caller-saved register

Case 4: reassignment of a return value register

For reassigning a return value registers to a callee-saved register, the code insert is similar to that of an argument register except the place of code insertion. If the function has to return a value, an instruction needs to be inserted to move the return value from the reassigned register to the return value register. Another case is that for each function call in the function, if the called function has return value, it has to move the return value from the return value register to the reassigned register. Figure 3-11 shows the example of the reassignment of a return value register to a callee-saved register.

For reassigning a return value register to a caller-saved register, the code insertion policy is the same as we depicted in the previous paragraph. There is one thing different: if the return

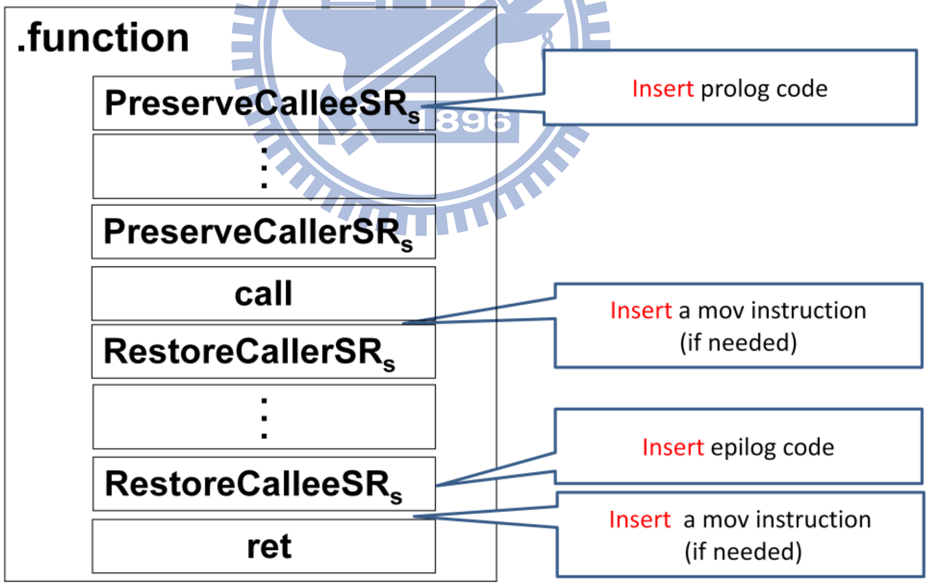


Figure 3-11 An example of reassignment a return value register to a callee-saved register

value register preserved/retrieved before/after each function call, it has to modify the stack offset to make sure that the value is spilled to the correct stack place. Figure 3-12 shows an example of reassignment a return value register to a caller-saved register. The summary of the reassignment overheads of all possible cases is shown in Table 3-1.

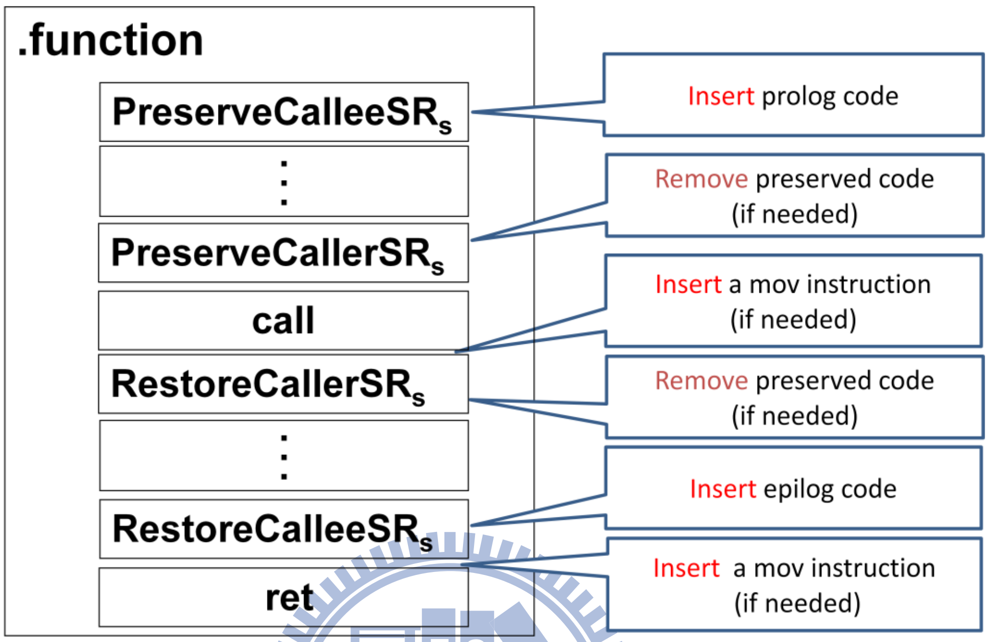

.function
PreserveCalleeSRs
PreserveCallerSRs
call
RestoreCallerSRs
RestoreCalleeSRs
ret
Insert prolog code
Remove preserved code (if needed)
Insert a mov instruction (if needed)
Remove preserved code (if needed)
Insert epilog code
Insert a mov instruction (if needed)

Figure 3-12 An example of reassignment a return value register to a caller-saved register

Table 3-1 Summary of reassignment overheads

After RRA / Before RRA	Callee-saved Register	Caller-saved Register
Callee-saved Register	1.May modify stack offset	1.Remove prologue/epilogue 2.Insert preserved Caller 3.Insert retrieved Caller
Temporary Register	1.May insert prologue/epilogue 2.May remove preserved/retrieved code	1.May modify stack offset
Argument Register	1.May insert prologue/epilogue 2.May remove preserved/retrieved code 3.May insert move instruction before each function call 4.May insert move instruction after prolog	1.Modify stack offset 2.May insert move instruction before each function call 3.May insert move instruction after prolog
Return Value Register	1.Insert prologue/epilogue 2.May remove preserved/retrieved code 3.May insert move instruction after each function call 4.May insert move instruction before function return	1.Modify stack offset 2.May insert move instruction after each function call 3.May insert move instruction before function return

According to the cases analysis described above, the equationof the reassignment overheads is expressed as follows: Note that each overhead is all computed in advanced.

$$\mathrm{Reassigment_{ovehead}} = \sum_{\mathrm{R_{bi}} \in \mathrm{RegUsed}} \sum_{\mathrm{R_{aj}} \in \mathrm{RegisterSet_A}}$$

$$\big(\mathrm{RM}[\mathrm{R_{bi}}, \mathrm{R_{aj}}] \times \mathrm{isCalleeToCallee}(\mathrm{R_{bi}}, \mathrm{R_{aj}}) \times \mathrm{CostofCalleeToCallee} +$$

$$\mathrm{RM}[\mathrm{R_{bi}}, \mathrm{R_{aj}}] \times \mathrm{isCalleeToCaller}(\mathrm{R_{bi}}, \mathrm{R_{aj}}) \times \mathrm{CostofCalleeToCaller} +$$

$$\mathrm{RM}[\mathrm{R_{bi}}, \mathrm{R_{aj}}] \times \mathrm{isTempToCallee}(\mathrm{R_{bi}}, \mathrm{R_{aj}}) \times \mathrm{CostofTempToCallee} +$$

$$\mathrm{RM}[\mathrm{R_{bi}}, \mathrm{R_{aj}}] \times \mathrm{isTempToCaller}(\mathrm{R_{bi}}, \mathrm{R_{aj}}) \times \mathrm{CostofTempToCaller} +$$

$$\mathrm{RM}[\mathrm{R_{bi}}, \mathrm{R_{aj}}] \times \mathrm{isArgToCallee}(\mathrm{R_{bi}}, \mathrm{R_{aj}}) \times \mathrm{CostofArgToCallee} +$$

$$\mathrm{RM}[\mathrm{R_{bi}}, \mathrm{R_{aj}}] \times \mathrm{isArgToCaller}(\mathrm{R_{bi}}, \mathrm{R_{aj}}) \times \mathrm{CostofArgToCaller} +$$

$$\mathrm{RM}[\mathrm{R_{bi}}, \mathrm{R_{aj}}] \times \mathrm{isRetToCaller}(\mathrm{R_{bi}}, \mathrm{R_{aj}}) \times \mathrm{CostofRetToCaller} +$$

$$\mathrm{RM}[\mathrm{R_{bi}}, \mathrm{R_{aj}}] \times \mathrm{isRetToCallee}(\mathrm{R_{bi}}, \mathrm{R_{aj}}) \times \mathrm{CostofRetToCallee}\big)$$

3.4.2 Constraints

We describe the constraints for the method 1 in this subsection.

1. Registers usage constraints:

$$\forall\ \mathrm{R_{bi}} \in \mathrm{RegUsed}, \quad \sum_{\mathrm{R_{aj}} \in \mathrm{RegsiterSet_A}} \mathrm{RM}[\mathrm{R_{bi}}, \mathrm{R_{aj}}] = 1 \qquad (2)$$

If a register $\mathrm{R_{bi}}$ is used before reassignment, it should be reassigned to one and only one register $\mathrm{R_{aj}}$ after reassignment.

$$\forall\ \mathrm{R_{bi}} \in \mathrm{RegNoUsed}, \quad \sum_{\mathrm{R_{aj}} \in \mathrm{RegsiterSet_A}} \mathrm{RM}[\mathrm{R_{bi}}, \mathrm{R_{aj}}] = 0 \qquad (3)$$

If a register $\mathrm{R_{bi}}$ is not used before reassignment, it should not occupy the register after reassignment but it can be reassigned by other registers.

2. Register reassignment constraints

$$\forall\ R_{aj} \in RegisterSet_{A,} \qquad \sum_{R_{bi} \in RegsiterSet_{A}} RM[R_{bi}, R_{aj}] \leq 1 \qquad (4)$$

For each register R_{aj} after reassignment, we restrict that it is reassigned by one register.

To clearly illustrate the constraints, the following example demonstrates how to use the reassignment matrix to build the constraints. Assume that a fictitious architecture only has four registers and two kinds of instruction widths, long and short formats. The long format instruction can access all the registers and the short format can only access the first three registers. The *RegisterSet_A* = {\$r1, \$r2, \$r3, \$r4} and *Registerset_S* = {\$r1, \$r2, \$r3}. Give a function that uses register \$r1, \$r2, \$r4. In this case, *RegUse* = {\$r1, \$r2, \$r4} and *RegNotUse* = {\$r3}. The constraints corresponding to reassignment matrix are shown in Figure 3-13.

	R_{a1}	R_{a2}	R_{a3}	R_{a4}	
R_{b1}					$\sum RM[R_{b1,}R_{ai}] = 1$
R_{b2}					$\sum RM[R_{b2,}R_{ai}] = 1$
R_{b3}					$\sum RM[R_{b3,}R_{ai}] = 0$
R_{b4}					$\sum RM[R_{b4,}R_{ai}] = 1$
	$\sum RM[R_{bi}R_{a1}] \leq 1$	$\sum RM[R_{bi,}R_{a2}] \leq 1$	$\sum RM[R_{bi,}R_{a3}] \leq 1$	$\sum RM[R_{bi,}R_{a4}] \leq 1$	

Figure 3-13 An example of the reassignment matrix

3. Instruction operands constraints

As mentioned in 3.4.1, these two constraints are added to linearize the objective function.

For each Inst $\in$ *U_INS*,

$$\mathrm{Inst} - \mathrm{isSmallReg}\big(\mathrm{InstTable}(\mathrm{Inst}, k)\big) \leq 0, 1 \leq k \leq \mathrm{RegOpndCount}(\mathrm{Inst}) \ (5)$$

$$\sum_{k=1}^{\mathrm{RegOpndCount(Inst)}} \mathrm{isSmallReg}(\mathrm{Inst\,Table}(\mathrm{Inst}, k)) - \mathrm{Inst} \leq \ \mathrm{RegOpndCount}(\mathrm{Inst}) - 1 \ (6)$$

4. Reassigning to small set register constraints

This constrains check whether the register after reassignment belongs to the small set register.

$\forall$, Reg $\in$ RegUsed

$$\mathrm{isSmallReg}(\mathrm{Reg}) \ - \sum_{R_{aj} \in \mathrm{RegsiterSet}_S} \mathrm{RM}[\mathrm{Reg}, R_{aj}] = 0 \qquad (7)$$

3.4.3 Summary

Finally, we give a brief summary of our objective function and constraints

Objective function

Maximize CSR

$$\mathrm{CSR} = \mathrm{CR}_{\mathrm{short}} - \ \mathrm{Reassignment}_{\mathrm{Overheads}}$$

$$\mathrm{CR}_{\mathrm{short}} = \sum_{\mathrm{Inst} \in U\_INS} \mathrm{Inst} \times \mathrm{SavedSize}$$

$$\mathrm{Reassigment}_{\mathrm{ovehead}} = \sum_{R_{bi} \in \mathrm{RegUsed}} \sum_{R_{aj} \in \mathrm{RegisterSet}_A}$$

$$\big(\mathrm{RM}[R_{bi}, R_{aj}] \times \mathrm{isCalleeToCallee}(R_{bi}, R_{aj}) \times \mathrm{CostofCalleeToCallee} \ +$$

$$\mathrm{RM}[R_{bi}, R_{aj}] \times \mathrm{isCalleeToCaller}(R_{bi}, R_{aj}) \times \mathrm{CostofCalleeToCaller} \ +$$

$$\mathrm{RM}[\mathrm{R_{bi}}, \mathrm{R_{aj}}] \times \mathrm{isTempToCallee}(\mathrm{R_{bi}}, \mathrm{R_{aj}}) \times \mathrm{CostofTempToCallee} +$$

$$\mathrm{RM}[\mathrm{R_{bi}}, \mathrm{R_{aj}}] \times \mathrm{isTempToCaller}(\mathrm{R_{bi}}, \mathrm{R_{aj}}) \times \mathrm{CostofTempToCaller} +$$

$$\mathrm{RM}[\mathrm{R_{bi}}, \mathrm{R_{aj}}] \times \mathrm{isArgToCallee}(\mathrm{R_{bi}}, \mathrm{R_{aj}}) \times \mathrm{CostofArgToCallee} +$$

$$\mathrm{RM}[\mathrm{R_{bi}}, \mathrm{R_{aj}}] \times \mathrm{isArgToCaller}(\mathrm{R_{bi}}, \mathrm{R_{aj}}) \times \mathrm{CostofArgToCaller} +$$

$$\mathrm{RM}[\mathrm{R_{bi}}, \mathrm{R_{aj}}] \times \mathrm{isRetToCaller}(\mathrm{R_{bi}}, \mathrm{R_{aj}}) \times \mathrm{CostofRetToCaller} +$$

$$\mathrm{RM}[\mathrm{R_{bi}}, \mathrm{R_{aj}}] \times \mathrm{isRetToCallee}(\mathrm{R_{bi}}, \mathrm{R_{aj}}) \times \mathrm{CostofRetToCallee})$$

Constraints

1. Register usage constraints:

$$\forall\, \mathrm{R_{bi}} \in \mathit{RegUsed}, \quad \sum_{\mathrm{R_{aj}} \in \mathrm{RegsiterSet_A}} \mathrm{RM}[\mathrm{R_{bi}}, \mathrm{R_{aj}}] = 1$$

$$\forall\, \mathrm{R_{bi}} \in \mathit{RegNoUsed}, \quad \sum_{\mathrm{R_{aj}} \in \mathrm{RegsiterSet_A}} \mathrm{RM}[\mathrm{R_{bi}}, \mathrm{R_{aj}}] = 0$$

2. Register reassignment constraints:

$$\forall\, \mathrm{R_{bi}} \in \mathit{RegisterSet_A}, \quad \sum_{\mathrm{R_{aj}} \in \mathrm{RegsiterSet_A}} \mathrm{RM}[\mathrm{R_{bi}}, \mathrm{R_{aj}}] \le 1$$

3. Instruction operands constraints:

$$\mathrm{Inst} - \ \mathrm{isSmallReg}(\mathrm{InstTable}(\mathrm{Inst}, \mathrm{k})) \le 0$$

$$\sum_{\mathrm{k}=1}^{\mathrm{RegOpndCount(Inst)}} \mathrm{isSmallReg}(\mathrm{InstTable}(\mathrm{Inst}, \mathrm{k})) - \mathrm{Inst} \le \mathrm{RegOpndCount}(\mathrm{Inst}) - 1$$

4. Reassigning to small set register constraints:

$$\forall\, \mathrm{Reg} \in \mathrm{RegUsed}, \mathrm{isSmallReg}(\mathrm{Reg}) \ - \sum_{\mathrm{R_{aj}} \in \mathrm{RegsiterSet_S}} \mathrm{RM}[\mathrm{Reg}, \mathrm{R_{aj}}] = 0$$

3.5 Example

We give an example to illustrate our register reassignment method by ILP in this section. We input a simple MIPS assembly code in Figure 3-14 and we will use this example to explain how to build ILP model for register reassignment. Note that the funcA takes one parameter as input.

```
 1    lw    $4   4($sp)
 2    lw    $8   8($sp)
 3   add    $9   $4     $8
 4   sub   $10   $9     $4
 5  addi    $4   $10    2
 6    lw   $25   %got(funcA)($gp)
 7  jalr   $25
 8    lw   $11   4($sp)
 9  andi   $12   $11    5
10  addi   $13   $12    1
11   ori    $2   $13    -1
12    jr   $31
```

Figure 3-14 An MIPS 32 assembly code

Here we use MIPS as our target architecture. The sets related to MIPS architecture are defined as follows:

$RegisterSet_A$ = {\$1, \$2, …, \$25}

$RegisterSet_S$ = {\$1, \$2, …, \$7}

CalleeSavedRegs = {\$16, \$17, …, \$23}

CallerSavedRegs = {\$2, \$3, …, \$15, \$24, \$25}

ArgRegs = {\$4, \$5, \$6, \$7}

RetRegs = {\$2, \$3}

TempRegs = {\$8, \$9, …, \$15, \$24, \$25}

$\text{InstSize}_{\text{long}}$ = 4 bytes

$\text{InstSize}_{\text{short}}$ = 2 bytes

SavedSize = 2 bytes

We analyze the input assembly code and define the sets related to the function features listed as follows:

U_INS = {Inst1, Inst2, Inst3, Inst4, Inst5, Inst8, Inst9, Inst10, Inst11}

S_INS = {Inst7, Inst12}

L_INS= {Inst6}

$\text{InstSize}_{\text{long}}$ = 4 bytes

$\text{InstSize}_{\text{short}}$ = 2 bytes

$\text{Saved}_{\text{Size}}$ = 2 bytes

RegUsed = {\$2, \$4, \$8, \$9, \$10, \$11, \$12, \$13, \$25}

RegNoUsed = {\$1, \$3, \$5, \$6, \$r7, \$14, \$15, \$16, \$17,…, \$24}

Reg, Reg $\in$ $\textit{RegsisterSet}_A$: register variable

X_{i_j}, i, j $\in$ $\textit{RegsisterSet}_A$: X_{i_j}=1 if register i is reassigned to register j

InstTable of *U_INS* is defined in Table 3-3.

Table 3-3 The contents of InstTable

Inst #	Opcode	RegOpnd Def	RegOpnd Use 1	RegOpnd Use 2
Inst1	lw	\$4		
Inst2	lw	\$8		
Inst3	add	\$9	\$4	\$8
Inst4	sub	\$10	\$9	\$4
Inst5	addi	\$4	\$10	
Inst8	lw	\$11		
Inst9	andi	\$12	\$11	
Inst10	addi	\$13	\$12	
Inst11	ori	\$2	\$13	

The reassignment overheads of used registers are listed in Table 3-4 and are all constants. \$4 and \$2 are used for passing parameter and receiving return value according to MIPS calling convention. If \$4 is reassigned, a move instruction should be inserted before function call. Therefore, it leads overheads. If \$2 is reassigned, a move instruction should be inserted before function return. Note that a register reassigned to itself has no overheads.

Table 3-4 Example of reassignment overheads

Register	Reassign to Caller-saved registers	Reassign to Callee-saved registers
\$2	2 bytes	10 bytes
\$4	2 bytes	10 bytes
\$8	0 bytes	8 bytes
\$9	0 bytes	8 bytes
\$10	0 bytes	8 bytes
\$11	0 bytes	8 bytes
\$12	0 bytes	8 bytes
\$13	0 bytes	8 bytes
\$25	0 bytes	8 bytes

3.5.1 Objective Function

Maximize$2*Inst1+2*Inst2+2*Inst3+2*Inst4+2*Inst5+2*Inst8+2*Inst9+2*Inst10+$

$2*Inst11 - 2*(X_{2\_1}+X_{2\_3}+X_{2\_4}+\cdots+X_{2\_15}+X_{2\_24}+X_{2\_25}) - 10*(X_{2\_16}+X_{2\_17}+\cdots+X_{2\_23})$

$- 2*(X_{4\_1}+X_{4\_2}+X_{4\_3}+X_{4\_5}+\cdots+X_{4\_15}+X_{4\_24}+X_{4\_25}) - 10*(X_{4\_16}+X_{4\_17}+\cdots+X_{4\_23}) - 8*$

$(X_{8\_16}+X_{8\_17}+\cdots+X_{8\_23}) - 8*(X_{9\_16}+X_{9\_17}+\cdots+X_{9\_23}) - 8*(X_{10\_16}+X_{10\_17}+\cdots+X_{10\_23}) - 8*$

$(X_{11\_16}+X_{11\_17}+\cdots+X_{11\_23}) - 8*(X_{12\_16}+X_{12\_17}+\cdots+X_{12\_23}) - 8*(X_{12\_16}+X_{12\_17}+\cdots+$

$X_{12\_23}) - 8*(X_{13\_16}+X_{13\_17}+\cdots+X_{13\_23}) - 8*(X_{25\_16}+X_{25\_17}+\cdots+X_{25\_23})$

3.5.2 Register Usage Constraints

$X_{2\_1}+X_{2\_2}+X_{2\_3}+X_{2\_4}+X_{2\_5}+\cdots+X_{2\_24}+X_{2\_25} = 1$

$X_{4\_1}+X_{4\_2}+X_{4\_3}+X_{4\_4}+X_{4\_5}+\cdots+X_{4\_24}+X_{4\_25} = 1$

$X_{8\_1}+X_{8\_2}+X_{8\_3}+X_{8\_4}+X_{8\_5}+\cdots+X_{8\_24}+X_{8\_25} = 1$

$X_{9\_1}+X_{9\_2}+X_{9\_3}+X_{9\_4}+X_{9\_5}+\cdots+X_{9\_24}+X_{9\_25} = 1$

$X_{10\_1}+X_{10\_2}+X_{10\_3}+X_{10\_4}+X_{10\_5}+\cdots+X_{10\_24}+X_{10\_25} = 1$

$X_{11\_1}+X_{11\_2}+X_{11\_3}+X_{11\_4}+X_{11\_5}+\cdots+X_{11\_24}+X_{11\_25} = 1$

$X_{12\_1}+X_{12\_2}+X_{12\_3}+X_{12\_4}+X_{12\_5}+\cdots+X_{12\_24}+X_{12\_25} = 1$

$X_{13\_1}+X_{13\_2}+X_{13\_3}+X_{13\_4}+X_{13\_5}+\cdots+X_{13\_24}+X_{13\_25} = 1$

$X_{25\_1}+X_{25\_2}+X_{25\_3}+X_{25\_4}+X_{25\_5}+\cdots+X_{25\_24}+X_{25\_25} = 1$

3.5.3 Register Reassignment Constraints

$X_{1\_1}+X_{2\_1}+X_{3\_1}+\cdots+X_{24\_1}+X_{25\_1} \leq 1$

$X_{1\_2}+X_{2\_2}+X_{3\_2}+\cdots+X_{24\_2}+X_{25\_2} \leq 1$

$X_{1\_3}+X_{2\_3}+X_{3\_3}+\cdots+X_{24\_3}+X_{25\_3} \leq 1$

$X_{1\_4}+X_{2\_4}+X_{3\_4}+\cdots+X_{24\_4}+X_{25\_4} \leq 1$

.

.

.

$X_{1\_24}+X_{2\_24}+X_{3\_24}+\cdots+X_{24\_24}+X_{25\_24} \leq 1$

$X_{1\_25}+X_{2\_25}+X_{3\_25}+\cdots+X_{24\_25}+X_{25\_25} \leq 1$

3.5.4 Instruction Operands Constraints

$\text{Inst1} - \text{Reg4} \leq 0$

$\text{Reg4} - \text{Inst1} \leq 0$

$\text{Inst2} - \text{Reg8} \leq 0$

$Reg8 - Inst2 \leq 0$

$Inst3 - Reg9 \leq 0$

$Inst3 - Reg4 \leq 0$

$Inst3 - Reg8 \leq 0$

$Reg9 + Reg4 + Reg8 - Inst3 \leq 2$

$Inst4 - Reg10 \leq 0$

$Inst4 - Reg9 \leq 0$

$Inst4 - Reg4 \leq 0$

$Reg10 + Reg9 + Reg4 - Inst4 \leq 2$

$Inst5 - Reg4 \leq 0$

$Inst5 - Reg10 \leq 0$

$Reg4 + Reg10 - Inst5 \leq 1$

$Inst8 - Reg11 \leq 0$

$Reg8 - Inst8 \leq 0$

$Inst9 - Reg12 \leq 0$

$Inst9 - Reg11 \leq 0$

$Reg12 + Reg11 - Inst9 \leq 1$

$Inst10 - Reg13 \leq 0$

$Inst10 - Reg12 \leq 0$

$\mathrm{Reg13} + \mathrm{Reg12} - \mathrm{Inst10} \le 1$

$\mathrm{Inst11} - \mathrm{Reg2} \le 0$

$\mathrm{Inst11} - \mathrm{Reg13} \le 0$

$\mathrm{Reg2} + \mathrm{Reg13} - \mathrm{Inst11} \le 2$

3.5.5 Reassigning to Small Set Register Constraints

$\mathrm{Reg1} - X_{1\_1} - X_{1\_2} - X_{1\_3} - X_{1\_4} - X_{1\_5} - X_{1\_6} - X_{1\_7} = 0$

$\mathrm{Reg2} - X_{2\_1} - X_{2\_2} - X_{2\_3} - X_{2\_4} - X_{2\_5} - X_{2\_6} - X_{2\_7} = 0$

$\mathrm{Reg3} - X_{3\_1} - X_{3\_2} - X_{3\_3} - X_{3\_4} - X_{3\_5} - X_{3\_6} - X_{3\_7} = 0$

$\mathrm{Reg4} - X_{4\_1} - X_{4\_2} - X_{4\_3} - X_{4\_4} - X_{4\_5} - X_{4\_6} - X_{4\_7} = 0$

·

·

·

$\mathrm{Reg24} - X_{24\_1} - X_{24\_2} - X_{24\_3} - X_{24\_4} - X_{24\_5} - X_{24\_6} - X_{24\_7} = 0$

$\mathrm{Reg25} - X_{25\_1} - X_{25\_2} - X_{25\_3} - X_{25\_4} - X_{25\_5} - X_{25\_6} - X_{25\_7} = 0$

We input this example to ILP solver, the final result is that we can save 14 bytes after register reassignment. Except $2, all the other registers belong to *RegisterSet*$_S$ after register reassignment. The final result is listed as follows:

- $X_{2\_2} = 1, X_{4\_4} = 1, X_{8\_7} = 1, X_{9\_6} = 1, X_{10\_10} = 1, X_{11\_5} = 1, X_{12\_3} = 1, X_{13\_1} = 1, X_{25\_25} = 1$

Except instruction 4 and 5, the input instructions all become S-format instructions after register reassignment. Figure 3-15 shows the final result.

```
 1   lw    $4   4($sp)
 2   lw    $7   8($sp)
 3   add   $6   $4       $7
 4   sub   $10  $6       $4
 5   addi  $4   $10      2
 6   lw    $25  %got(funcA)($gp)
 7   jalr  $25
 8   lw    $5   4($sp)
 9   andi  $3   $5       5
10   addi  $1   $3       1
11   ori   $2   $1       -1
12   jr    $31
```

Figure 3-15 Register reassignment results for example 3.5

Chapter 4 ILP Formulation for Register Reassignment Method Ⅱ

In this chapter, we will present an extended method which relaxes the register reassignment constraint of the method 1 mentioned in Chapter 3. In the method 1, each register used in a function is restricted to be reassigned to exactly one register. In this extended method, if the live ranges of the registers do not overlap, they can be reassigned to a same register. Therefore, the liveness information of the physical registers should be collected for register reassignment. For computing liveness, we use the web, which is a set of uses and definitions of physical registers in the statements of the function [20]. Each physical register may have several webs. We apply the backward data-flow analysis to build the def-use chain and merge the def-use chains with same use to form the webs. After building the webs, we need to analyze the interference between the webs. Two webs are called interfered if one is live in the definition of another. We use this rule to build the interference constraints to prevent the interfered webs to be reassigned to the same register. In Section 4.1, we first give the notation used in the method 2. In Section 4.2 we will present the ILP formulation for this method. Finally, we will use an example to illustrate this method.

4.1 Notation

Before we present the ILP formulation for method 2, we first define the notation which will be used in this model.

4.1.1 ISA Features

The following sets, variables and functions are the same with the Subsection 3.3.1. We listed it as follows:

- $RegisterSet_A$: set of all the physical registers except special purpose registers in the target architecture.
- $RegisterSet_S$: the set of all the physical registers which can be indexed by S-format instructions.
- isSmallReg(W), W $\in$ *Webs*: a boolean variable function which represent whether the given web will be reassigned to $RegisterSet_S$.

The following variables are related to ISA format.

- $\text{InstSize}_{\text{long}}$: the number of bytes of a long instruction.
- $\text{InstSize}_{\text{Short}}$: the number of bytes of a short instruction.
- SavedSize: the number of saved bytes when a long instruction is translated to a short instruction.

The following sets are related to the calling convention.

- *CalleeSavedRegs*: the set of the physical registers saved by callee.
- *CallerSavedRegs*: the set of the physical registers saved by caller.
- *TempRegs*: the set of physical registers which are used for storing temporary value.

- *ArgRegs*: the set of physical registers which are used for passing arguments.
- *RetRegs*: the set of physical registers which are used for passing return values.

The following sets defined for the classification of input instructions.

- *L_INS*: the set of certain L-format instructions. An instruction which has no equivalent short format instruction or has oversized immediate value for the equivalent short format instruction belongs to this category.
- *S_INS*: the set of certain S-format instructions. An instruction which has an equivalent short format instruction by arbitrary register assignment belongs to this set.
- *U_INS*: the set of instructions each of which may be either L-format or S-format according to the register assignment. This kind of instructions is the input of our register reassignment method.

4.1.2 Function Features

We make the definitions of three new sets for the webs and list as follows:

- *Webs*: the set of all webs in a function.
- $WebsReg_i$, $i \in U\_INS$: the set of register operands of instruction i.
- $WebsInterference_{i\_j}$, $i \in RegUsed$, j is the set of webs of register i: the set of webs which interfere with the web, W_{i_j}.

The following notations are the same with Subsection 3.3.2

- *RegUsed*: the set of registers used in the function.

- RegOpndCount(Inst), Inst ∊ *U_INS*: a function which returns the number of register operands of uncertain format instruction Inst.
- Inst, Inst∊*U_INS*: instruction variable of each *U_INS*. Inst = 1 when Inst can be converted to S-format instruction after reassignment.
- InstTable(Inst, k), Inst ∊ *U_INS*; $1 \leq k \leq$ RegOpndCount(Inst): a function that returns the web number of kth register number used by Inst.

For the callee-saved registers used in a function after register reassignment, we define the variables as follows:

- CalleeUseReg, Reg ∊ *CalleeSavedRegs*: variables for checking whether callee-saved registers are used after reassignment.
- CalleeNotUseReg, Reg ∊ *CalleeSavedRegs*: variables for checking whether callee-saved registers are not used after reassignment.

4.1.3 Reassignment Matrix

In 3.4.2, we have shown the structure of reassignment matrix (RM). Each row and column of RM represents the register number. In Method 2, we modify this RM such that each row of RM stands for the webs of a function. We defined RM and variables used in Method 2 as follows:

- $X_{i\_j\_k}$, i ∊ *RegUsed*, j ∊ the set of webs of register i, k ∊ $RegisterSet_A$: each element of the RM. $X_{i\_j\_k}$ = 1 if the jth web of register i is reassigned to register k.

- W_{i_j}, i ∈ *RegisterSet*$_A$, j ∈ the set of webs of register i: the jth web of register i before reassignment. $W_{i\_j} = 1$ if the jth web of register will be reassigned to the small set registers after reassignment.

4.2 Formulation

In this section, we present the formulation of the second ILP model. First, we show the objective function and constraints of the model.

4.2.1 Objective Function

Our objective function is to maximize the code size reduction: CSR, when all the constraints are satisfied. CSR is comprised of two parts: CR_{short} and $Reassignment_{Overheads}$.

$$\textbf{Maximize}\quad \mathrm{CSR} \qquad (1)$$

$$\mathrm{CSR} = \mathrm{CR_{short}} - \mathrm{Reassignment_{Overheads}} - \mathrm{ProEpilog_{Callee}}$$

1. CR_{short}

CR_{short} is the saved size by translating L-format instruction to S-format instruction.

$$\mathrm{CR_{short}} = \sum_{\mathrm{Inst} \in U\_INS} \mathrm{Inst} \times \mathrm{SavedSize}$$

2. $Reassignment_{Overheads}$

$$\mathrm{Reassigment_{ovehead}} = \sum_{\mathrm{i \in RegUsed}} \sum_{\mathrm{j \in WebsReg_i}} \sum_{\mathrm{Reg \in RegisterSet_A}} ($$

$$\mathrm{RM}[\mathrm{W_{i\_j}}, \mathrm{Reg}] \times \mathrm{isCalleeToCallee}(\mathrm{W_{i\_j}}, \mathrm{Reg}) \times \mathrm{CostofCalleeToCallee} \; +$$

$$\mathrm{RM}[\mathrm{W_{i\_j}}, \mathrm{Reg}] \times \mathrm{isCalleeToCaller}(\mathrm{W_{i\_j}}, \mathrm{Reg}) \times \mathrm{CostofCalleeToCaller} \; +$$

$$\mathrm{RM}[\mathrm{W_{i\_j}}, \mathrm{Reg}] \times \mathrm{isTempToCallee}(\mathrm{W_{i\_j}}, \mathrm{Reg}) \times \mathrm{CostofTempToCallee} +$$

$$\mathrm{RM}[\mathrm{W}_{\mathrm{i\_j}}, \mathrm{Reg}] \times \mathrm{isTempToCaller}(\mathrm{W}_{\mathrm{i\_j}}, \mathrm{Reg}) \times \mathrm{CostofTempToCaller} +$$

$$\mathrm{RM}[\mathrm{W}_{\mathrm{i\_j}}, \mathrm{Reg}] \times \mathrm{isArgToCallee}(\mathrm{W}_{\mathrm{i\_j}}, \mathrm{Reg}) \times \mathrm{CostofArgToCallee} +$$

$$\mathrm{RM}[\mathrm{W}_{\mathrm{i\_j}}, \mathrm{Reg}] \times \mathrm{isArgToCaller}(\mathrm{W}_{\mathrm{i\_j}}, \mathrm{Reg}) \times \mathrm{CostofArgToCaller} +$$

$$\mathrm{RM}[\mathrm{W}_{\mathrm{i\_j}}, \mathrm{Reg}] \times \mathrm{isRetToCaller}(\mathrm{W}_{\mathrm{i\_j}}, \mathrm{Reg}) \times \mathrm{CostofRetToCaller} +$$

$$\mathrm{RM}[\mathrm{W}_{\mathrm{i\_j}}, \mathrm{Reg}] \times \mathrm{isRetToCallee}(\mathrm{W}_{\mathrm{i\_j}}, \mathrm{Reg}) \times \mathrm{CostofRetToCallee})$$

3. ProEpilogCalleeSR: we need to check whether callee-saved registers are used after reassignment. There are four cases for callee-saved registers to determine the code insertion at prolog and epilog. Table 4-1 presents the code insertion for callee-saved registers at prologue and epilogue.

$$\forall\ \mathrm{Reg} \in \mathrm{CalleeSavedRegs}, \mathrm{ProEpilog}_{\mathrm{calleeSR}}(\mathrm{Reg}) =$$
$$\mathrm{CalleeUse}_{\mathrm{Reg}} \times \mathrm{CostofUseCallee}_{\mathrm{Reg}} +$$
$$\mathrm{CalleeNotUse}_{\mathrm{Reg}} \times \mathrm{CostofNotUseCallee}_{\mathrm{Reg}}$$

Table 4-1 Instruction insertions for callee-saved registers at prologue/epilogue after register reassignment

After Reassignment / Before Reassignment	Used	NotUsed
Used	No Insertion	Remove Prologue/Epilogue
NotUsed	Insert Prologue/Epilogue	No Insertion

4.2.2 Constraints

In Subsection 3.4.2, we have shown the constraints for method one. For the method two, we relax the register reassignment constraints such that each register can be reassigned more than once. Meanwhile, we also set a new constraint for the interfered webs. The constraints are listed as follows:

1. Webs usage constraints

For $W_{i\_j} \in \textit{Webs}$, W_{i_j} should be reassigned to one and only one register R_j

$$\forall\, W_{i\_j} \in \text{Webs}, \sum_{R_j \in RegisterSet_A} RM[W_{i\_j}, R_j] = 1 \tag{2}$$

2. Register reassignment constraints

For each Reg $\in$ *RegisterSet*$_A$, Reg can be reassigned any times by W_{i_j}

$$\forall\, \text{Reg} \in \text{RegisterSet}_A, \sum_{W_i \in Webs} RM[W_{i\_j}, \text{Reg}] \geq 0 \tag{3}$$

3. Webs interference constraints

For $W_{a\_b} \in \textit{WebsInterference}_{i\_j}$, W_{a_b} and W_{i_j} cannot reassigned to the same register

$$\forall\, \text{Reg} \in \text{RegisterSet}_A, RM[W_{a\_b}, \text{Reg}] + RM[W_{i\_j}, \text{Reg}] \leq 1 \tag{4}$$

4. Instruction operand constraints

For each Inst $\in$ *U_INS*,

$$\text{Inst} - \text{isSmallReg}(\text{InstTable}(\text{Inst}, k)) \leq 0, 1 \leq k \leq \text{RegOpndCount}(\text{Inst}) \tag{5}$$

$$\sum_{k=1}^{\text{RegOpndCount(Inst)}} \text{isSmallReg}(\text{Inst Table}(\text{Inst}, k)) - \text{Inst} \leq \text{RegOpndCount}(\text{Inst}) - 1 \tag{6}$$

5. Callee-saved registers overheads constraints: these constraints check whether each callee-saved register has been reassigned.

$\forall$ i $\in$ RegUsed, j $\in$ $\mathrm{Webs_i}$, Reg $\in$ CalleeSavedRegs

$$\mathrm{CalleeUse_{Reg}} - \mathrm{X_{i,j,Reg}} \geq 0 \tag{7}$$

$$\left(\sum_{\mathrm{Reg \in CalleeSavedRegs}} \mathrm{X_{i,j,Reg}}\right) - \mathrm{CalleeUse_{Reg}} \geq 0 \tag{8}$$

$$\mathrm{CalleeUse_{Reg}} + \mathrm{CalleeNotUse_{Reg}} = 1 \tag{9}$$

6. Reassigning to small set register constraints

$\forall\ \mathrm{W_{i\_j}} \in \mathit{Web},$

$$\mathrm{isSmallReg}(\mathrm{W_{i\_j}}) - \sum_{\mathrm{Reg_{AR} \in RegisterSet_s}} \mathrm{RM}[\mathrm{W_{i\_j}}, \mathrm{Reg_{AR}}] = 0 \tag{10}$$

4.3 Example

We give an example to illustrate our register reassignment method by ILP in this subsection. Here we use the same example defined in Section 3.5. Here we also use MIPS as our target architecture; therefore, the target-dependent set would not be defined again.

The following sets are related to the input function.

U_INS = {Inst1, Inst2, Inst3, Inst4, Inst5, Inst8, Inst9, Inst10, Inst11}

S_INS = {Inst7, Inst12}

L_INS= {Inst6}

RegUsed = {\$2, \$4, \$8, \$9, \$10, \$11, \$12, \$13, \$25}

RegNoUsed = {\$1, \$3, \$5, \$6, \$r7, \$14, \$15, \$16, \$17,…, \$24}

Webs = {W_{2_1}, W_{4_1}, W_{4_2},W_{8_1}, W_{9_1}, W_{10_1}, W_{11_1}, W_{12_1}, W_{13_1}}

$\mathit{WebsInterference}_{2\_1} = \{W_{13\_1}\}$

$\mathit{WebsInterference}_{4\_1} = \{\ \}$

$\mathit{WebsInterference}_{4\_2} = \{W_{10\_1}\}$

$\mathit{WebsInterference}_{8\_1} = \{W_{4\_1}\}$

$\mathit{WebsInterference}_{9\_1} = \{W_{4\_1}, W_{8\_1}\}$

$\mathit{WebsInterference}_{10\_1} = \{W_{9\_1}, W_{4\_1}\}$

$\mathit{WebsInterference}_{11\_1} = \{\ \}$

$\mathit{WebsInterference}_{12\_1} = \{W_{11\_1}\}$

$\mathit{WebsInterference}_{13\_1} = \{W_{12\_1}\}$

$\mathit{WebsInterference}_{25\_1} = \{W_{4\_2}\}$

W_{i_j}, $W_{i\_j} \in \mathit{Webs}$: webs variable.

X_{i_j}, $i \in \mathit{Webs}$, $j \in \mathit{RegsisterSet}_A$: $X_{i\_j}=1$ if Web i reassigns to register j.

$\mathrm{CalleeUse}_{\mathrm{Reg}}$, $\mathrm{Reg} \in \mathit{CalleeSavedRegs}$: $\mathrm{CalleeUse}_{\mathrm{Reg}} = 1$ if Reg is used after reassignment.

$\mathrm{CalleeNotUse}_{\mathrm{Reg}}$, $\mathrm{Reg} \in \mathit{CalleeSavedRegs}$: $\mathrm{CalleeNotUse}_{\mathrm{Reg}} = 1$ if Reg is not used after reassignment.

InstTable of *U_INS* is listed in Table 4-2.

Table 4-2 The contents of InstTable

Inst #	Opcode	RegOpnd Def	RegOpnd Use 1	RegOpnd Use 2
Inst1	lw	$4		
Inst2	lw	$8		
Inst3	add	$9	$4	$8
Inst4	sub	$10	$9	$4
Inst5	addi	$4	$10	
Inst8	lw	$11		
Inst9	andi	$12	$11	
Inst10	addi	$13	$12	
Inst11	ori	$2	$13	

Assume that the reassignment overheads of each used register are listed in Table 4-3. C1,C2,.., and C16 are the reassignment overheads of the corresponding webs and are all constants. Note that a web reassigned to itself has no overheads.

Table 4-3 Reassignment overheads of each web

Web	Reassign to Caller-saved registers	Reassign to Callee-saved registers
W_{2_1}	2 bytes	2 bytes
W_{4_1}	0 bytes	0 bytes
W_{4_2}	2 bytes	2 bytes
W_{8_1}	0 bytes	0 bytes
W_{9_1}	0 bytes	0 bytes
W_{10_1}	0 bytes	0 bytes
W_{11_1}	0 bytes	0 bytes
W_{12_1}	0 bytes	0 bytes
W_{13_1}	0 bytes	0 bytes
W_{25_1}	0 bytes	0 bytes

4.3.1 Objective Function

Maximize $2*Inst1+2*Inst2+2*Inst3+2*Inst4+2*Inst5+2*Inst8+2*Inst9 +2*Inst10$

$+2*Inst11 - 2*(X_{2\_1\_1}+X_{2\_1\_3}+\ldots+X_{2\_1\_15}+X_{2\_1\_24}+X_{2\_1\_25}) - 2 * (X_{2\_1\_16}+X_{2\_1\_17}+\ldots$

$+X_{2\_1\_23}) - 2*(X_{4\_2\_1}+X_{4\_2\_2}+\ldots+X_{4\_2\_15}+X_{4\_2\_24}+X_{4\_2\_25}) - 2*(X_{4\_2\_16}+X_{4\_2\_17}+\ldots$

$+X_{4\_2\_23}) - 8 * CalleeUse_{16} - 8 * CalleeUse_{17} - 8 * CalleeUse_{18} - 8 * CalleeUse_{19} -8 *$

$CalleeUse_{20} - 8 * CalleeUse_{21} - 8 * CalleeUse_{22} - 8 * CalleeUse_{23}$

4.3.2 Web Usage Constraints

$X_{2\_1\_1}+X_{2\_1\_2}+X_{2\_1\_3}+X_{2\_1\_4}+X_{2\_1\_5}+\ldots+X_{2\_1\_24}+X_{2\_1\_25} = 1$

$X_{4\_1\_1}+X_{4\_1\_2}+X_{4\_1\_3}+X_{4\_1\_4}+X_{4\_1\_5}+\ldots+X_{4\_1\_24}+X_{4\_1\_25} = 1$

$X_{4\_2\_1}+X_{4\_2\_2}+X_{4\_2\_3}+X_{4\_2\_4}+X_{4\_2\_5}+\ldots+X_{4\_2\_24}+X_{4\_2\_25} = 1$

$X_{8\_1\_1}+X_{8\_1\_2}+X_{8\_1\_3}+X_{8\_1\_4}+X_{8\_1\_5}+\ldots+X_{8\_1\_24}+X_{8\_1\_25} = 1$

$X_{9\_1\_1}+X_{9\_1\_2}+X_{9\_1\_3}+X_{9\_1\_4}+X_{9\_1\_5}+\ldots+X_{9\_1\_24}+X_{9\_1\_25} = 1$

$X_{10\_1\_1}+X_{10\_1\_2}+X_{10\_1\_3}+X_{10\_1\_4}+X_{10\_1\_5}+\ldots+X_{10\_1\_24}+X_{10\_1\_25} = 1$

$X_{11\_1\_1}+X_{11\_1\_2}+X_{11\_1\_3}+X_{11\_1\_4}+X_{11\_1\_5}+\ldots+X_{11\_1\_24}+X_{11\_1\_25} = 1$

$X_{12\_1\_1}+X_{12\_1\_2}+X_{12\_1\_3}+X_{12\_1\_4}+X_{12\_1\_5}+\ldots+X_{12\_1\_24}+X_{12\_1\_25} = 1$

$X_{13\_1\_1}+X_{13\_1\_2}+X_{13\_1\_3}+X_{13\_1\_4}+X_{13\_1\_5}+\ldots+X_{13\_1\_24}+X_{13\_1\_25} = 1$

$X_{25\_1\_1}+X_{25\_1\_2}+X_{25\_1\_3}+X_{25\_1\_4}+X_{25\_1\_5}+\ldots+X_{25\_1\_24}+X_{25\_1\_25} = 1$

4.3.3 Register Reassignment Constraints

$X_{2\_1\_1}+X_{4\_1\_1}+X_{4\_2\_1}+X_{8\_1\_1}+ X_{9\_1\_1}+ X_{10\_1\_1}+ X_{11\_1\_1} + X_{12\_1\_1}+ X_{13\_1\_1} \geq 0$

$X_{2\_1\_2}+X_{4\_1\_2}+X_{4\_2\_2}+X_{8\_1\_2}+ X_{9\_1\_2}+ X_{10\_1\_2}+ X_{11\_1\_2} + X_{12\_1\_2}+ X_{13\_1\_2} \geq 0$

$X_{2\_1\_3}+X_{4\_1\_3}+X_{4\_2\_3}+X_{8\_1\_3}+ X_{9\_1\_3}+ X_{10\_1\_3}+ X_{11\_1\_3} + X_{12\_1\_3}+ X_{13\_1\_3} \geq 0$

$X_{2\_1\_4}+X_{4\_1\_4}+X_{4\_2\_4}+X_{8\_1\_4}+ X_{9\_1\_4}+ X_{10\_1\_4}+ X_{11\_1\_4} + X_{12\_1\_4}+ X_{13\_1\_4} \geq 0$

.

.

.

$X_{2\_1\_24}+X_{4\_1\_24}+X_{4\_2\_24}+X_{8\_1\_24}+ X_{9\_1\_24}+ X_{10\_1\_24}+ X_{11\_1\_24} + X_{12\_1\_24}+ X_{13\_1\_24} \geq 0$

$X_{2\_1\_24}+X_{4\_1\_24}+X_{4\_2\_25}+X_{8\_1\_24}+ X_{9\_1\_24}+ X_{10\_1\_24}+ X_{11\_1\_24} + X_{12\_1\_24}+ X_{13\_1\_24} \geq 0$

4.3.4 Web Interference Constraints

1. Webs interfered with W_{2_1}

$X_{2\_1\_1}+X_{13\_1\_1} \leq 1$

$X_{2\_1\_2}+X_{13\_1\_2} \leq 1$

.

.

$X_{2\_1\_24}+X_{13\_1\_24} \leq 1$

$X_{2\_1\_25}+X_{13\_1\_25} \leq 1$

2. Webs interfered with W_{4_2}

$X_{4\_2\_1}+X_{10\_1\_1} \leq 1$

$X_{4\_2\_2}+X_{10\_1\_2} \leq 1$

.

.

$X_{4\_2\_24}+X_{10\_1\_24} \leq 1$

$X_{4\_2\_25}+X_{10\_1\_25} \leq 1$

3. Webs interfered with W_{8_1}

$X_{8\_1\_1}+X_{4\_1\_1} \leq 1$

$X_{8\_1\_2}+X_{4\_1\_2} \leq 1$

.

.

.

$X_{8\_1\_24}+X_{4\_1\_24} \leq 1$

$X_{8\_1\_25}+X_{4\_1\_25} \leq 1$

4. Webs interfered with W_{9_1}

$X_{9\_1\_1}+X_{4\_1\_1} \leq 1$

$X_{9\_1\_2}+X_{4\_1\_2} \leq 1$

.

.

.

$X_{9\_1\_24}+X_{4\_1\_24} \leq 1$

$X_{9\_1\_25}+X_{4\_1\_25} \leq 1$

$X_{9\_1\_1}+X_{8\_1\_1} \leq 1$

$X_{9\_1\_2}+X_{8\_1\_2} \leq 1$

.

.

.

$X_{9\_1\_24}+X_{8\_1\_24} \leq 1$

$X_{9\_1\_25}+X_{8\_1\_25} \leq 1$

5. Webs interfered with W_{10_1}

$X_{10\_1\_1}+X_{9\_1\_1} \leq 1$

$X_{10\_1\_2}+X_{9\_1\_2} \leq 1$

.

.

.

$X_{10\_1\_24}+X_{9\_1\_24} \leq 1$

$X_{10\_1\_25}+X_{9\_1\_25} \leq 1$

$X_{10\_1\_1}+X_{4\_1\_1} \leq 1$

$X_{10\_1\_2}+X_{4\_1\_2} \leq 1$

.

.

.

$X_{10\_1\_24}+X_{4\_1\_24} \leq 1$

$X_{10\_1\_25}+X_{4\_1\_25} \leq 1$

6. Webs interfered with W_{12_1}

$X_{12\_1\_1}+X_{11\_1\_1} \leq 1$

$X_{12\_1\_2}+X_{11\_1\_2} \leq 1$

.

.

.

$X_{12\_1\_24}+X_{11\_1\_24} \leq 1$

$X_{12\_1\_25}+X_{11\_1\_25} \leq 1$

7. Webs interfered with W_{13_1}

$X_{13\_1\_1}+X_{12\_1\_1} \leq 1$

$X_{13\_1\_2}+X_{12\_1\_2} \leq 1$

.

.

.

$X_{13\_1\_24}+X_{12\_1\_24} \leq 1$

$X_{13\_1\_25}+X_{12\_1\_25} \leq 1$

8. Webs interfered with W_{25_1}

$X_{25\_1\_1}+X_{4\_2\_1} \leq 1$

$X_{25\_1\_2}+X_{4\_2\_2} \leq 1$

.

.

.

$X_{25\_1\_24}+X_{4\_2\_24} \leq 1$

$X_{25\_1\_25}+X_{4\_2\_25} \leq 1$

4.3.5 Instruction Operands Constraints

$\text{Inst1} - W_{4\_1} \leq 0$

$W_{4\_1} - \text{Inst1} \leq 0$

$\text{Inst2} - W_{8\_1} \leq 0$

$W_{8\_1} - \text{Inst2} \leq 0$

$\text{Inst3} - W_{9\_1} \leq 0$

$\text{Inst3} - W_{4\_1} \leq 0$

$\text{Inst3} - W_{8\_1} \leq 0$

$W_{9\_1}+W_{4\_1} + W_{8\_1} - \text{Inst3} \leq 2$

$\text{Inst4} - W_{10\_1} \leq 0$

$\text{Inst4} - W_{9\_1} \leq 0$

$\text{Inst4} - W_{4\_1} \leq 0$

$W_{10\_1}+W_{9\_1} + W_{4\_1} - \text{Inst4} \leq 2$

$\text{Inst5} - W_{4\_2} \leq 0$

$\text{Inst5} - W_{10\_1} \leq 0$

$W_{4\_2}+W_{10\_1} - \text{Inst5} \leq 1$

$\text{Inst8} - W_{11\_1} \leq 0$

$W_{11\_1} - \text{Inst8} \leq 0$

$\text{Inst9} - W_{12\_1} \leq 0$

$\text{Inst9} - W_{11\_1} \leq 0$

$W_{12\_1} + W_{11\_1} - \text{Inst9} \leq 1$

$\text{Inst10} - W_{13\_1} \leq 0$

$\text{Inst10} - W_{12\_1} \leq 0$

$W_{13\_1} + W_{12\_1} - \text{Inst10} \leq 1$

$\text{Inst11} - W_{2\_1} \leq 0$

$\text{Inst11} - W_{13\_1} \leq 0$

$W_{2\_1} + W_{13\_1} - \text{Inst11} \leq 0$

4.3.6 Reassigning to Small Set Register Constraints

$W_{2\_1} - X_{2\_1\_1} - X_{2\_1\_2} - X_{2\_1\_3} - X_{2\_1\_4} - X_{2\_1\_5} - X_{2\_1\_6} - X_{2\_1\_7} = 0$

$W_{4\_1} - X_{4\_1\_1} - X_{4\_1\_2} - X_{4\_1\_3} - X_{4\_1\_4} - X_{4\_1\_5} - X_{4\_1\_6} - X_{4\_1\_7} = 0$

$W_{4\_2} - X_{4\_2\_1} - X_{4\_2\_2} - X_{4\_2\_3} - X_{4\_2\_4} - X_{4\_2\_5} - X_{4\_2\_6} - X_{44\_2\_7} = 0$

$W_{8\_1} - X_{8\_1\_1} - X_{8\_1\_2} - X_{8\_1\_3} - X_{8\_1\_4} - X_{8\_1\_5} - X_{8\_1\_6} - X_{8\_1\_7} = 0$

$W_{9\_1} - X_{9\_1\_1} - X_{9\_1\_2} - X_{9\_1\_3} - X_{9\_1\_4} - X_{9\_1\_5} - X_{9\_1\_6} - X_{9\_1\_7} = 0$

$W_{10\_1} - X_{10\_1\_1} - X_{10\_1\_2} - X_{10\_1\_3} - X_{10\_1\_4} - X_{10\_1\_5} - X_{10\_1\_6} - X_{10\_1\_7} = 0$

$W_{11\_1} - X_{11\_1\_1} - X_{11\_1\_2} - X_{11\_1\_3} - X_{11\_1\_4} - X_{11\_1\_5} - X_{11\_1\_6} - X_{11\_1\_7} = 0$

$W_{12\_1} - X_{12\_1\_1} - X_{12\_1\_2} - X_{12\_1\_3} - X_{12\_1\_4} - X_{12\_1\_5} - X_{12\_1\_6} - X_{12\_1\_7} = 0$

$W_{13\_1} - X_{13\_1\_1} - X_{13\_1\_2} - X_{13\_1\_3} - X_{13\_1\_4} - X_{13\_1\_5} - X_{13\_1\_6} - X_{13\_1\_7} = 0$

$W_{25\_1} - X_{25\_1\_1} - X_{25\_1\_2} - X_{25\_1\_3} - X_{25\_1\_4} - X_{25\_1\_5} - X_{25\_1\_6} - X_{25\_1\_7} = 0$

4.3.7 Callee-Saved Registers Overheads Constraints

$X_{2\_1\_16} + CalleeUse_{16} \geq 0$

$X_{4\_1\_16} + CalleeUse_{16} \geq 0$

$X_{4\_2\_16} + CalleeUse_{16} \geq 0$

$X_{8\_1\_16} + CalleeUse_{16} \geq 0$

$X_{9\_1\_16} + CalleeUse_{16} \geq 0$

$X_{10\_1\_16} + CalleeUse_{16} \geq 0$

$X_{11\_1\_16} + CalleeUse_{16} \geq 0$

$X_{12\_1\_16} + CalleeUse_{16} \geq 0$

$X_{13\_1\_16} + CalleeUse_{16} \geq 0$

$X_{25\_1\_16} + CalleeUse_{16} \geq 0$

$X_{2\_1\_16} + X_{4\_1\_16} + X_{4\_2\_16} + X_{8\_1\_16} + X_{9\_1\_16} + X_{10\_1\_16} + X_{11\_1\_16} + X_{12\_1\_16} + X_{13\_1\_16} +$

$X_{25\_1\_16} - CalleeUse_{16} \geq 0$

$CalleeUse_{16} + CalleeNotUse_{16} = 1$

.

.

.

$X_{2\_1\_23} + CalleeUse_{23} \geq 0$

$X_{4\_1\_23} + CalleeUse_{23} \geq 0$

$X_{4\_2\_23} + CalleeUse_{23} \geq 0$

$X_{8\_1\_23} + CalleeUse_{23} \geq 0$

$X_{9\_1\_23} + CalleeUse_{23} \geq 0$

$X_{10\_1\_23} + CalleeUse_{23} \geq 0$

$X_{11\_1\_23} + CalleeUse_{23} \geq 0$

$X_{12\_1\_23} + CalleeUse_{23} \geq 0$

$X_{13\_1\_23} + CalleeUse_{23} \geq 0$

$X_{25\_1\_23} + CalleeUse_{23} \geq 0$

$X_{2\_1\_23} + X_{4\_1\_23} + X_{4\_2\_23} + X_{8\_1\_23} + X_{9\_1\_23} + X_{10\_1\_23} + X_{11\_1\_23} + X_{12\_1\_23} + X_{13\_1\_23} +$

$X_{25\_1\_23} - CalleeUse_{23} \geq 0$

$CalleeUse_{23} + CalleeNotUse_{23} = 1$

Assume that all C1, C2, …, and C16 are all zero. We input this model to ILP solver, the final result is that we can save 18 bytes after register reassignment. All the webs belong to *RegisterSet*$_S$ after reassignment since method 1 only allows each register can be reassigned once. In the method 2, we only restrict interfered webs which cannot be reassigned to the same registers. Therefore, we can achieve better result compared with method 1. The result of the above example is listed as follows:

- $X_{2\_1\_2} = 1, X_{4\_1\_4} = 1, X_{4\_2\_4}, X_{8\_1\_2} = 1, X_{9\_1\_3} = 1, X_{10\_1\_5} = 1, X_{11\_1\_6} = 1, X_{12\_1\_4} = 1,$
  $X_{13\_1\_5} = 1, X_{25\_1\_25} = 1$

The result of register reassignment is listed in Figure 4-1.

```
 1    lw    $4    4($sp)
 2    lw    $2    8($sp)
 3   add    $3    $4       $2
 4   sub    $5    $9       $4
 5  addi    $4    $5       2
 6    lw   $25    %got(funcA)($gp)
 7  jalr   $25
 8    lw    $6    4($sp)
 9  andi    $4    $6       5
10  addi    $5    $4       1
11   ori    $2    $5       -1
12    jr   $31
```

Figure 4-1 Register reassignment results for example 4.3

Chapter 5 Experiments

In this chapter we present the experiment results. Firstly the experiment environment and ILP solver used in our model are introduced. Then, we will show the experiment results of the two ILP models. Finally we will make a discussion of the results and compare to the previous research.

5.1 Environment

The experiment environments include three major parts: ILP solver module, compiler back-end, and target architecture. In the following subsections, we will introduce the above three parts.

5.1.1 IBM ILOG CPLEX

IBM ILOG CPLEX is an optimization software for mathematical programming [21]. ILOG supports some fundamental algorithms to solve linear programming, mixed integer programming, quadratic programming, and quadratic constrained programming problems. It provides flexible interfaces that can interact with the existing programming languages. In this thesis, we use this tool to create and solve the integer linear programming model for register reassignment.

5.1.2 LLVM

We implement our design in back-end of LLVM[22]. LLVM is the abbreviation of Low Level Virtual Machine which is a compiler framework proposed by University of Illinois at Urbana-Champaign. It provides a very simple, hierarchical and modular back-end design such that it may easily be used to implement a new idea or design on it. It also supports a lot of popular architecture such x86, ARM, SPARC, MIPS and so on. LLVM provides many optimization options such as register coalescing, post register allocation scheduling and so on. In this thesis, we add a new phase after register allocation for register reassignment by ILP. After that, LLVM will call the instruction formatting to output the assembly file.

5.1.3 Target Architecture

The target ISA is MIPS-like which assumes that the long or short instruction mode is changed by a special instruction bit in the instruction rather than a specific mode switch instruction. MIPS defined 28 general purposed registers and 4 special purposed registers; $gp, $sp, $fp, and $ra. In general purposed registers, $r0 is hard-lined zero, $r26 and $r27 is preserved for kernel usage. Hence, these three registers and the four special purposed registers are not considered for register reassignment. In addition, MIPS also defined callee-saved, caller-saved, argument and return value registers. Therefore, several register sets defined for our model as listed as follows:

- $RegisterSet_S$ = {\$1, \$2, …, \$7},
- $RegisterSet_A$ = {\$1, \$2, …, \$25},
- *CalleeSavedRegs* = {\$16, \$17, …, \$23},
- *CallerSavedRegs* = {\$8, \$9, …, \$15, \$24, \$25},
- *ArgRegs* = {\$4, \$5, \$6, \$7},
- *RetArgs* = {\$2, \$3}
- *TempArgs* = {\$8, \$9, …, \$15, \$24, \$25}.

5.2 Benchmark Evaluation Results

In Figure 1-2, we have showed the distribution of the instruction type and the uncertain in which format instruction count ratio and our motivation also from that figure. In this subsection, we are going to present the translation rate of short instructions and compare it with the related work.

Figure 5-1 shows the short instruction translation rate between different methods Here the direct conversion means that the code is directly produced and formatted without the register reassignment. On the average, we translate more than 14.6 and 6.6% uncertainly in which format instructions to short instructions compared with direct conversion and related work in method one. For method two, we translate more than 34.4%, 26.5% and 20% uncertain in which format instructions to S-format instruction compared with direct conversion, Ku's method and method 1.

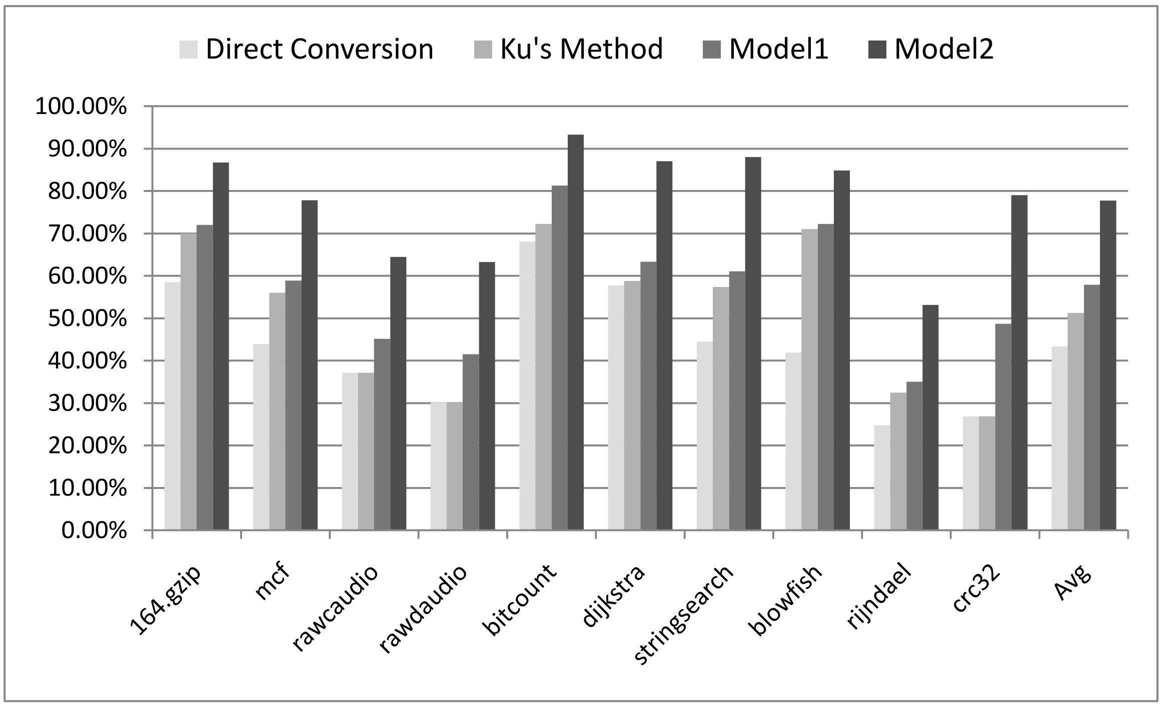


Figure 5-1 Translation rate of uncertain in which format instructions to short instructions

Figure 5-2 shows the additional instruction counts of our two methods and the related work. Ku’s method did not apply to the following three benchmarks: rawcuadio, rawdaudio and crc, since the code size reduction is less than that of the direct conversion. For some benchmarks such as crc and stringsearch, our method suffers from calling convention overheads significantly. However, the instruction counts of some other benchmarks like rawdaudio and bitcount may even less than that of the original program since we remove the prologue and epilogue for callee-saved registers after reassignment. For these programs which increase the instructions after reassignment such as stringsearch and blowfish, they may degrade the performance; but for some other benchmarks, they can reduce the total instruction counts. Therefore, register reassignment in some cases may not affect the performance; it depends on the characteristics of the program.

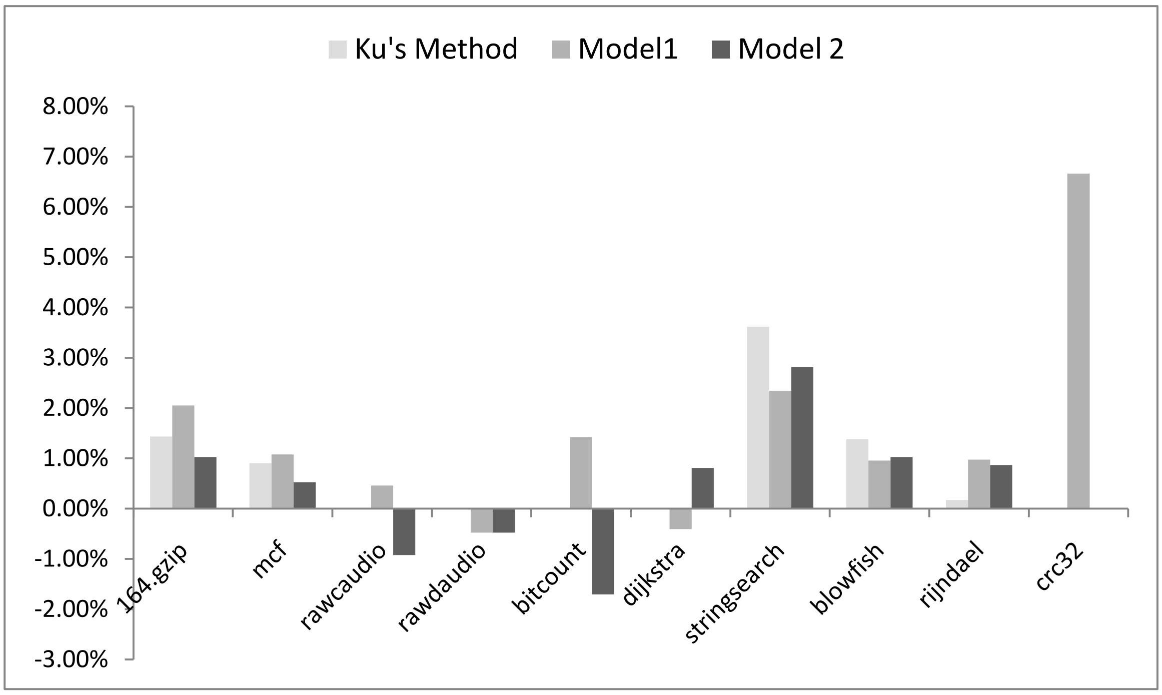


Figure 5-2 Additional instruction count comparison

Figure 5-3 shows the code size reduction ratios of our two methods compared with direct conversion, Ku's method and an ideal case in which all uncertain in which format instructions are assumed to be translated to short instructions. On the average, we reduce about 30.11% and 34.40% code size in contrast to the original program after applying our two methods, respectively. Our two different methods come up with different results. Method 2 can further improve the code size than method one since it allows that a register may be reassigned more than once. In next section, we will further analyze the impact of these two different methods.

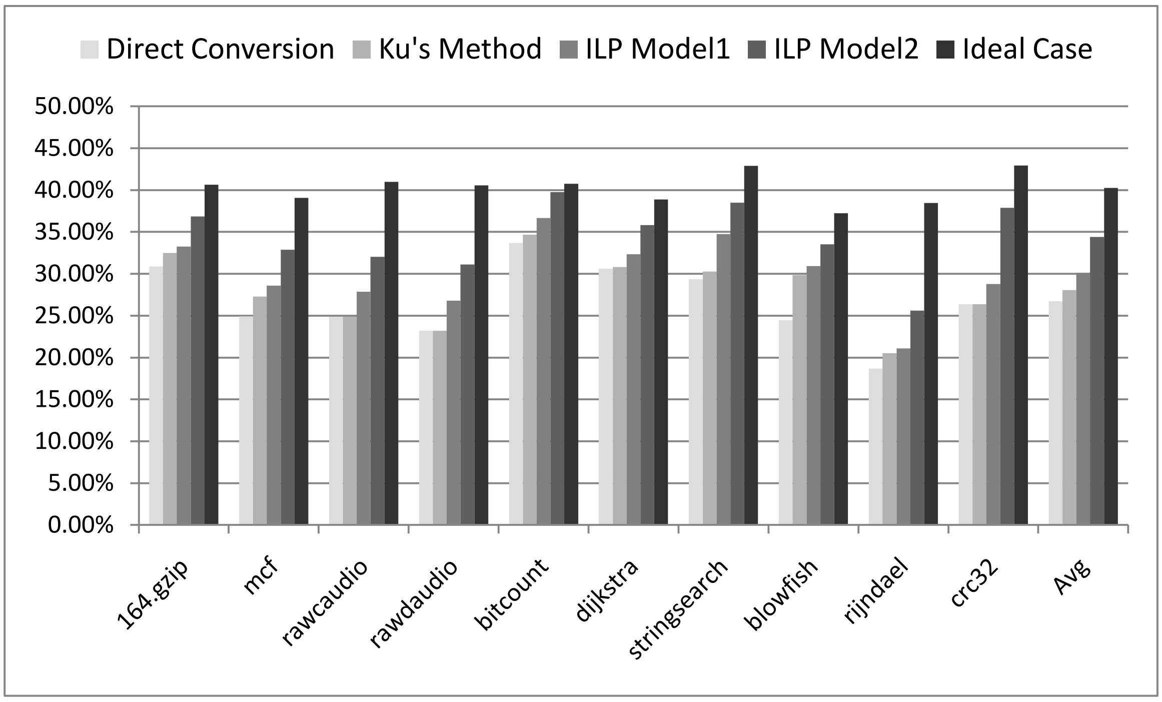


Figure 5-3 Code size reduction comparison

5.3 Experiments Discussion

In this section, we will discuss and analyze the results of our two register reassignment methods.

5.3.1 Discussion of Method Ⅰ

In Method 1, the translation rates of some programs have a large gap compared to the ideal case. In fact, the ideal case is a very ideal upper bound that only establishment by using small set registers for register assignment without any additional spill code. In other words, if the register pressure in a function is higher than the number of *RegisterSet*$_S$, this upper bound is difficult to be achieved.

We observe the number of instructions may be covered by the frequently occurred registers. By gathering the register occurrences of each instruction and sorting in descending order, we discover that the frequent occurred registers can cover all instructions of the functions in some programs. Here we infer that the translation rate may have positive correlation with the coverage ratio of frequent occurred registers.

In Method 1, each used register can only be reassigned to one register, i.e. ILP solver will map the registers with high instruction coverage ratio and lower calling convention overheads to reassign to the small set registers. If the function does not have many function calls, the translation rates of it may be dominated by the coverage rate of the frequent occurred registers. We analyze the benchmarks to judge whether the short instruction translation rates consistent with our inference. Since the number of small set registers is 7, we will observe the top 7 frequent occurred registers.

Figure 5-4 shows the unlzh function in 164.gizp. In this function, the top 7 frequent occurred registers cover about 91% instructions of the function. In fact, there are 85% instructions can be translated to short instructions after applied our method 1.

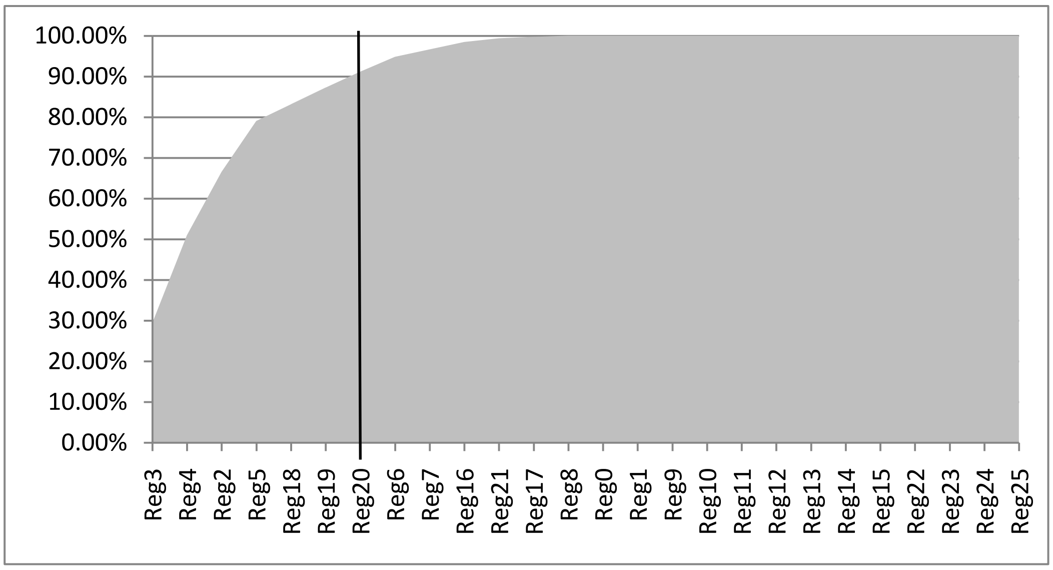


Figure 5-4 Instruction coverage ratios of top 7 frequent occurrence registers of unlzh

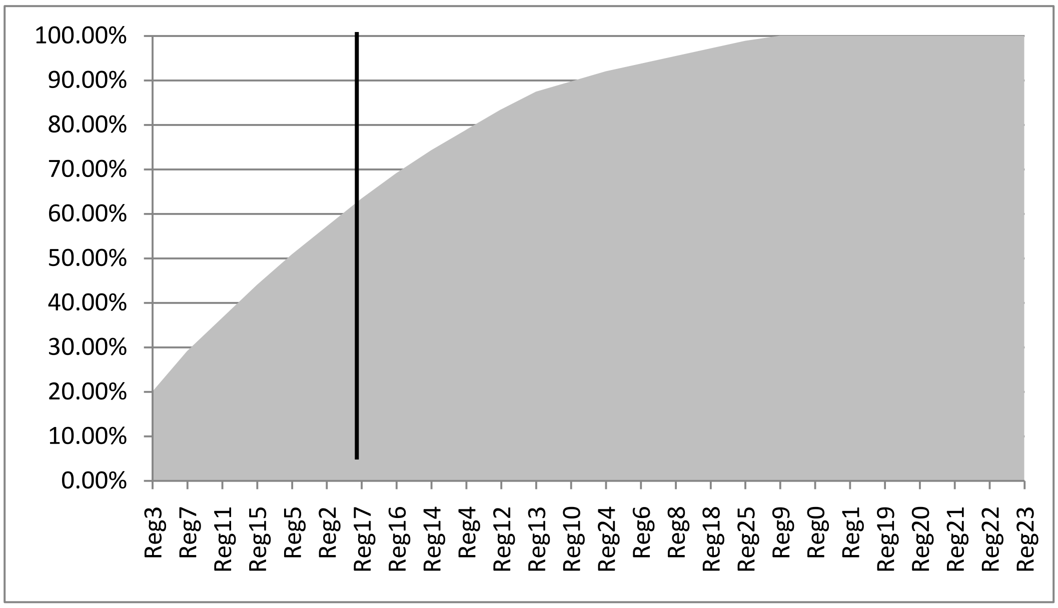


Figure 5-5 Instruction coverage ratios of the top 7 registers of Rawcaudio

Figure 5-5 shows the instruction coverage ratio of the top 7 frequent occurred registers in the main function of Rawcaudio. The top 7 registers can cover about 63% instructions of the function and the short instruction translation rate is about 45%.

Figure 5-6 shows the instruction coverage ratio of the top 7 frequent occurred registers in the main function of CRC. Although the top 7 registers can cover 75% instructions of the function, there are four callee-saved registers in the frequent occurred registers and many function calls, i.e. the ILP solver may not chose those registers mapping to the small set registers because of high reassignment overheads. The short instruction translation rate of CRC is about 49%.

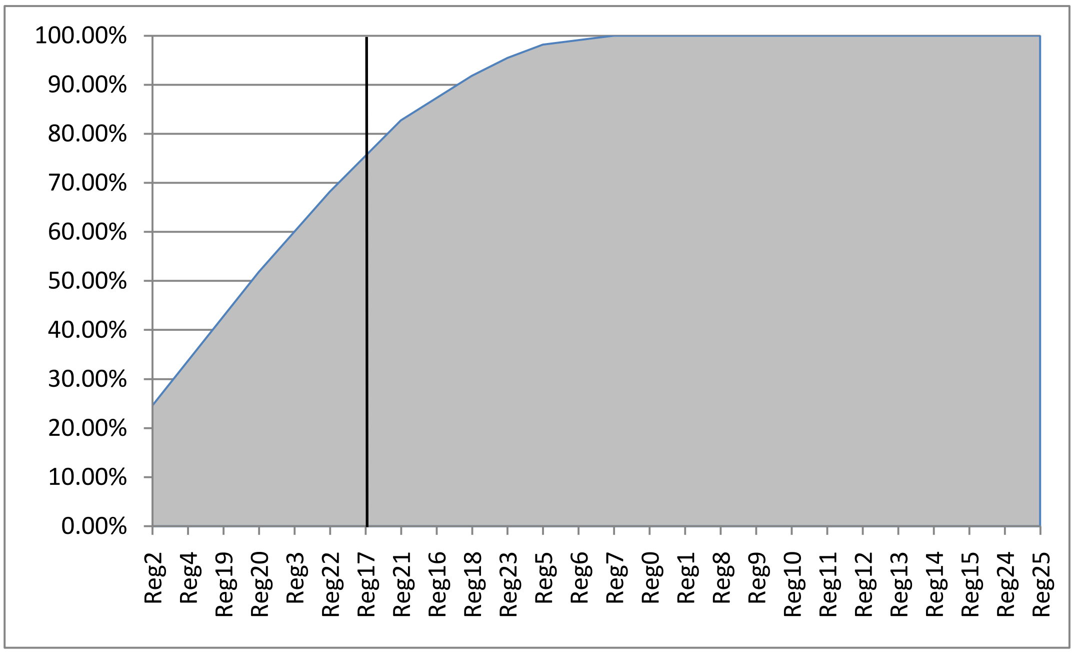


Figure 5-6 Instruction coverage ratios of the top 7 frequent occurrence registers in CRC

For Rijndael and Rawdaudio, they also have the same property we have mentioned above. Basically, our register reassignment policy prefers higher short instruction translation rate and lower calling convention overheads.

5.3.2 Discussion of Method Ⅱ

In the second method, we relax the restrictions to allow each register to be reassigned more than once. Obviously, the results are better than method one since more registers after reassignment can map to *RegisterSet*$_S$. Nevertheless, some programs still have a large gap compared to the upper bound, such as rawcaudio, rawdaudio, and rijndael. For rawcaudio and rawdaudio, we find some webs with long life time but low usage count. However, our method does not split the live range of this kind of webs, and we only restrict that the interference registers cannot be reassigned to the same register. On the other hand, rijndael suffers from high register pressures so that method 2 cannot work well on it. For these reasons, we don’t achieve good code size reduction on these programs.

Chapter 6 Conclusions and Future Works

Code size issue for a memory constrained system is as important as performance especially for versatile functionalities requirements. Mixed-width ISA is proposed to reduce the memory usage from the architecture point of view. There are many commercial processors for embedded systems have adopted mixed-width ISA.

We introduce register reassignment for code size reduction for two reasons: efficiency and complexity. The efficiency is that we can inherit existing compiler back-end paths and add a new phase for optimizing code size. For the complexity point of view, it is not easy to modify the existing register allocation algorithm for adapting the new features since the new features may conflict with the original goal of the register allocation.

In this thesis, we model register reassignment problem to be two different integer linear programs. In average, we reduce 30.11% and 34.40% code size in the first and second method by our experiment. Through this result, we can set an upper bound for this register reassignment problem under our assumptions and use this result to evaluate the results of all possible heuristic algorithms.

In the future, there are still some works to do for reducing code size by register reassignment using integer linear programming. In our method, we only consider the calling convention overheads but ignore the performance effect of the code inserted. If the code insertion is in the hotspot parts such as a loop, it may degrade the performance significantly.

Therefore, we can try to use the profiling results to find the hotspot of a function and avoid doing code insertion in the hotspot. However, the profiling results may not be accurate since they are collected statically. A conservative way to estimate performance impact of the place of code insertion can be also evaluated. We may treat each loop in a function as hotspot, and then set a special weight for the code insertion in such places.

Reference

[1] *Advanced RISC Machine Ltd.* *http://www.arm.com*.

[2] *MIPS Technology* *http://www.mips.com/*.

[3] *Andes Technology*. *Andes Instruction Set Architecuture Specification, 2008*. Available: http://www.andestech.com

[4] "microMIPS Instruction Set Architecture. Uncompromised Performance, Minimum System Cost. MD00690 Revision 01.00. Oct. 2009."

[5] R. Phelan, "Improving Arm Code Density and Performance -- New Thumb Extensions to the Arm Architecture. Arm Thumb-2 Core Technology Whitepaper," June, 2003 2003.

[6] A. Krishnaswamy and R. Gupta, "Mixed-width instruction sets," *Commun. ACM,* vol. 46, pp. 47-52, 2003.

[7] *The SPEC2000 Benchmark*, *http://www.spec.org/cpu2000/*.

[8] *MediaBench*, *http://euler.slu.edu/~fritts/mediabench/*.

[9] *MiBench*, *http://www.eecs.umich.edu/mibench/*.

[10] T.-Y. Yang, "Register Allocation of JIT Compiler for Mixed-Width ISA for Code Size Reduction," Master Thesis, Department of Computer Science, National Chiao-Tung University, HisnChu,Taiwan, R.O.C, 2009.

[11] J.-S. Wang, *et al.*, "Reducing Code Size by Graph Coloring Register Allocation and Assignment Algorithm for Mixed-Width ISA Processor," the Proceedings of the 2009 International Conference on Computational Science and Engineering, 2009, Volume 2,pp. 174-181.

[12] Bor-Yeh Shen, Wei-Chung Hsu, and Wuu Yang, "Register Reassignment for Mixed-Width ISAs is an NP-Complete Problem," the Proceedings of the International Multi-Conference on Complexity, Informatics and Cybernetics (IMCIC 2010), Orlando, Florida, USA, April 6-9, 2010, pp. 139-143.

[13] S. Lee, *et al.*, "Selective code transformation for dual instruction set processors," *ACM Trans. Embed. Comput. Syst.*, 2007 ,Vol. 6, p. 10,.

[14] A. Halambi, *et al.*, "An Efficient Compiler Technique for Code Size Reduction Using Reduced Bit-Width ISAs," the Proceedings of the conference on Design, automation and test in Europe (DATE), 2002, pp. 402-408.

[15] L. Xianhua, *et al.*, "Efficient code size reduction without performance loss," presented at the Proceedings of the 2007 ACM symposium on Applied computing (SAC), Seoul, Korea, 2007, pp.666-672.

[16] A. Krishnaswamy and R. Gupta, "Profile guided selection of ARM and thumb instructions," *SIGPLAN Not.* , 2002,vol. 37, pp. 56-64.

[17] T. J. K. E. v. Koch et al., "Integrated instruction selection and register allocation for compact code generation exploiting freeform mixing of 16- and 32-bit instructions," presented at the Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization (CGO 2010), Toronto, Ontario, Canada, 2010, pp. 180-189.

[18] Y.-L. Ku, "Code Size Reduction with Register Reassignment for Mixed-Width ISA Processsors," Master Thesis, Department of Computer Science, National Chiao Tung University, HsinChu, 2009.

[19] P. Barth, *Logic-based 0-1 constraint programming*: Kluwer Academic Publishers, 1996.

[20] S. S. Muchnick, *Advanced compiler design and implementation*: Morgan Kaufmann Publishers Inc., 1997.

[21] IBM. (2010, *ILOG CPLEX 12.2)* *http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/*.

[22] C. Lattner. *The LLVM Compiler Infrastructure.* http://llvm.org.