國 立 交 通 大 學

資訊科學與工程研究所

碩 士 論 文

在即時編譯器中針對混合固定長度指令集所設計的暫存器分配演算法以求縮減程式碼

Register Allocation of JIT Compiler for Mixed-Width ISA for Code Size Reduction

研 究 生：楊天元

指導教授：單智君　博士

中 華 民 國 九 十 八 年 十 二 月

在即時編譯器中針對混合固定長度指令集所設計的暫存器分配演算法以求縮減程式碼

Register Allocation of JIT Compiler for Mixed-Width ISA for Code Size Reduction

研 究 生：楊天元　　　　Student：Tian-Yuan Yang
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

December 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年十二月

在即時編譯器中針對混合固定長度指令集所設計的暫存器分配演算法以求縮減程式碼

研究生：楊天元　　　　　　　　　　　　　　　　　　　　指導教授：單智君

摘要

在嵌入式系統中儲存程式的記憶體是相當重要的資源，於是減少程式碼的大小成為了重要的議題。其中一種解決方法是用“混合寬度指令集架構”。一般而言，這樣的架構提供了兩種固定長度指令集，能夠執行一般指令(通常為 32 位元)以及短指令(通常為 16 位元)。在這樣的架構中程式可由兩種指令組合而成，如果程式使用了越多的短指令其程式碼大小就越小。然而短指令上有著一些限制，例如只能使用較短立即值範圍以及存取較少的暫存器，這些都影響了程式中短指令使用的比例多寡，其中如何分配可被短指令存取的暫存器影響最大。這篇論文中我們針對混合寬度指令集架構提出了一個適用於即時編譯器的快速暫存器演算法，這個演算法能得到 26%程式碼縮減。

Register Allocation of JIT Compiler for Mixed-Width ISA for Code Size Reduction

Student: Tian-Yuan Yang advisor：Dr. Jyh-Jiun Shann

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

In embedded system, memory is a precious resource, and that reducing program code size becomes an important issue. One promising approach for code reduction is employing “mixed-width instruction set architecture (ISA)”. This kind of architecture usually provides two fixed width instruction sets, a long instruction set (usually 32-bit) and a short instruction set (usually 16-bit). In other words, program can be compiled into native code by using these two kinds of instructions mixedly. Obviously, the more short instructions are used in the program, the smaller program will be. However, there are some limitations for using short instructions. For example, they can only encode smaller range of immediate value and a subset of whole architecture registers. The short of registers is the most important issue among all limitations. Consequently, a register allocation which can allocate register more appropriately is needed for mixed-width ISA. In this thesis we propose a fast specific register allocation algorithm for JIT compiler which is more suitable for mixed-width instruction set architecture than traditional ones and get about 26% code size reduction with neglectable performance loss.

致謝及序言

首先感謝我的指導老師 單智君教授，在我碩士生涯中對我細心教誨，不時的討論並指點我正確的方向，使我在這些年中獲益匪淺。老師對學問的嚴謹更是我輩學習的典範。同時，也感謝口試委員，楊武教授與雍忠教授，由於教授們的指導與建議，才使得此篇論文更加完整與充實。另外，也感謝實驗室的另一位老師，鍾崇斌教授，在每次的報告之中給予學生指導與建議。

感謝奕緯學長、裕生學長給予我的幫助，不僅僅是研究相關的建議與花費心力的討論，在平日生活相處上也給予我許多寶貴的經驗，讓我在完成學業之餘也豐富了我的人生。同時也謝謝實驗室的學長姐、同儕以及學弟妹，在我的碩士生涯中給予我許多幫助，與你們一同留下的回憶此生難以忘懷。

最後要謝謝我的家人與親友，你們的支持與陪伴，一直是我精神上最堅強的支柱。

楊天元 2009.12.01

Table of Contents

List of Figures

Chapter 1 Introduction

In recent years, RISC processors are increasingly being used to design embedded systems. Using such architecture in embedded system provides certain advantages, such as increased design flexibility, high computing power and low on-chip power consumption. However, RISC processor systems suffer from the problem of poor code density which may require more ROM for storing program code [1].

One promising approach for reducing code size is to employ "Mixed-width instruction set architecture (ISA)" where the processors support different fixed-width instruction sets [2]. In general, there are 32- and 16- bit width instruction sets in Mixed-width ISAs, called L-Format (32bits) and S-Format (16bits) instruction sets. In these ISAs, most L-Format instructions have equivalent S-Format instructions, which means that the same operation could be achieved by either L- or S-Format instruction. Therefore, the more S-Format instructions are used in program the smaller program will be. However, there are several limitations of S-Format instructions such as limited operations, fewer bits to index registers and hold immediate values. Due to these limitations, using S-Format instructions should be under estimation and consideration. Moreover, when processor switches execution mode between L-Format and S-Format, there must be a mechanism to specify the execution mode, called "mode switch". There are two type of switching mechanisms: "By mode switching instruction" and "By instruction encoding". In present, mixed-width ISA with mode-switch by instruction encoding is a trend so that our research targets this kind of architecture. More description of mixed-width ISA will be introduced in Chapter 2.

In consideration of variant platforms in embedded systems, portability is an important issue for embedded application development. High portability can decrease development cost. In this case, JAVA is a suitable programming language [3]. In JAVA platform, JAVA compiler is different from traditional compiler, such as C-compiler; it translates JAVA codes into bytecodes, an intermediate representation, rather than native codes. These bytecodes will be executed by execution engines of JAVA virtual machine (JVM) [4] which runs above underlying physical machine as shown in Figure 1-1.

There are two kinds of execution engines in JVM: interpreter which interprets and executes one bytecode at once; Just-In-Time compiler (JIT compiler) which translates a segment of bytecodes into native codes and then executes them. Interpreter has short interactive time but lower performance. In contrast, JIT compiler has better performance but need more startup time than interpreter. JIT compiler is a dynamic compiler for runtime environment, which is used during execution. Therefore, light weight and efficient optimizations (such as a fast register allocation) are important for JIT compiler to reduce compilation overhead.

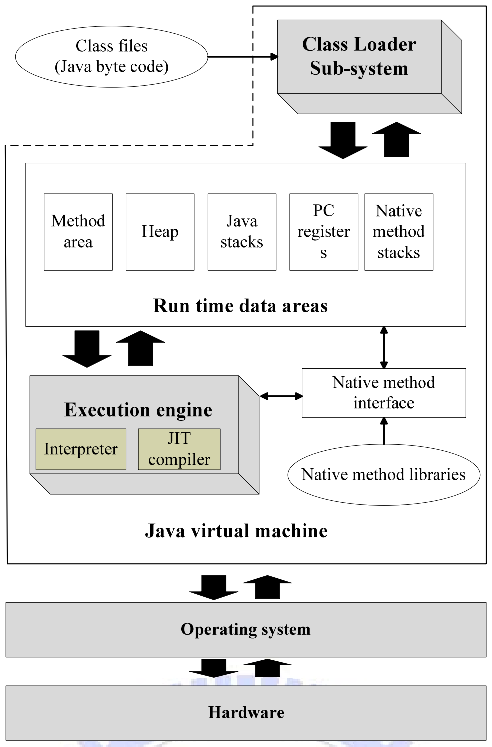


Figure 1-1: Java virtual machine (JVM)

Using RISC processor to design embedded system is a trend, but as what we mentioned before, it suffers from the problem of poor code density. Although there is an approach such as mixed-width ISA to ease this problem, it still needs some specific compiler optimizations to improve code density for mixed-width ISA. Moreover, these specific optimizations should be fast for JIT compiler because JIT compiler is important execution engine for improving application performance in JAVA and JAVA is a popular and appropriate programming language for embedded application development.

1.1 Research Motivation

So far we have introduced the mixed-width ISA: an approach to reduce code size by supporting L- and S-Format instruction sets. By observing the code produced by traditional compiler, we have discovered that even employing mixed-width ISA with mode switch by instruction encoding, most instructions cannot be formed in S-Format because of the limitations such as insufficient bits for indexing registers and holding immediate value, which means code size is not decreased much. As shown in Figure 1-2, there are only about 38 percents of instructions can be translated into S-Format and about 55 percents of instructions can't because their operands' register numbers are out of the range of the registers that can be indexed by S-Format instructions.

Therefore, if we can implement a particular register allocation for mixed-width ISA with mode-switch by instruction encoding, the code size reduction may be improved. Because we want to apply this design to JIT compiler to improve code density, the time complexity of the algorithm must be as low as possible.

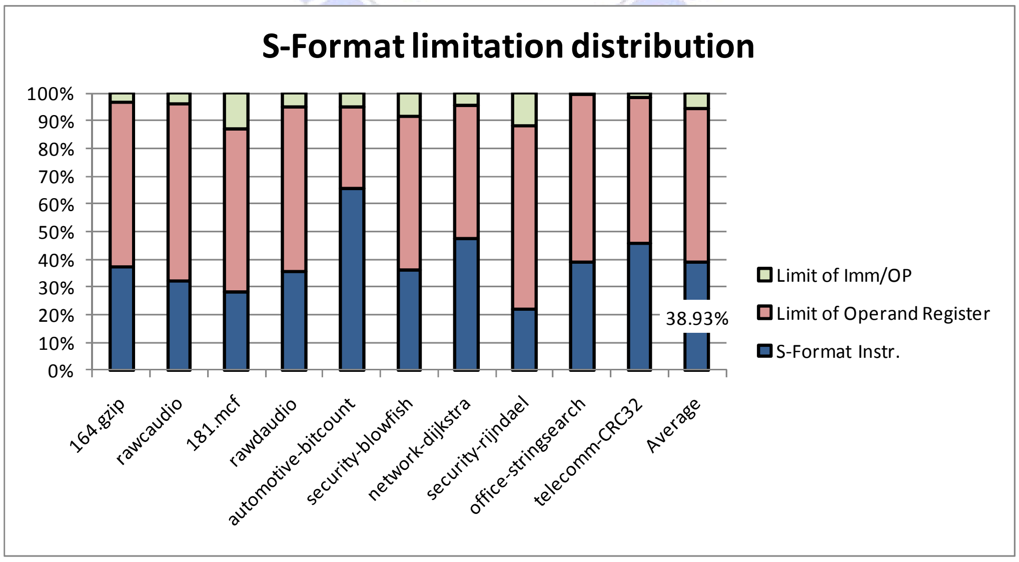


Figure 1-2: Distribution of limitations of S-Format instructions

1.2 Research Objective

To make more instructions to fit the limitation of operand registers, we propose a fast register allocation algorithm which takes consideration of S-Format limitations to improve code density. Here we name our design “Hierarchical Linear Scan Register Allocation” (HLSRA), since it allocates register hierarchically through one linear scanning. Along with that we want to apply HLSRA into the JIT compiler, there are two main goals to accomplish:

(1) Reducing code size

HLSRA will allocate registers according to a specific policy which is designed with consideration of register limitation of S-Format instruction. Therefore, the usage of registers will be more appropriate for mixed-with ISA and there will be more instructions meeting register limitation. Therefore, more instructions can be translate into S-Format instructions and code size reduction will be increased

(2) Low time complexity

In dynamic compiler, such as JIT compiler, compilation time makes a great impact on performance. Hence, our design has to be fast and efficient for such run-time environments. To make our design fast, we will implement it based on a known fast register allocation algorithm.

1.3 Organization of this Thesis

The remaining chapters of this thesis are organized as follows: In Chapter 2, we will provide background knowledge about mixed-width ISA and the register allocation algorithm which related to our research. In Chapter 3, we will introduce the basis of our design, and then describe the algorithm of “Hierarchical Linear Scan Register Allocation”. Chapter 4 will demonstrate the simulation environment and results of this work. And finally, Chapter 5, a summary and future work will be made and discussed.

Chapter 2 Background

In the first part of this chapter, we will describe more detail about the mixed-width ISA, including limitations of S-Format instruction, mechanism of mode switch and instruction alignment. In the second part, a fast traditional register allocation algorithm called “linear scan register allocation” will be introduced. At last, we will give a summary of this chapter.

2.1 Mixed-width ISA

Code size is an important issue in embedded system all the time, especially in RISC embedded processor which has suffered from the problem of poor code density. Consequently, mixed-width instruction set architectures (ISAs) have been proposed to make a good tradeoff between performance and code size. There are several mixed-width ISAs provided commercially, for examples, ARM’s ARM/Thumb ISA [5], MIPS’ MIPS32/MIPS16 ISA [6], Andes’ AndeStar ISA [7], and etc. For example, there are two different width instruction sets in MIPS’s architecture: the one called MIPS32 is a 32-bit width instruction set, and the other called MIPS16 is a 16-bit width instruction set. For convenience, we call them L- (32-bit) and S-Format (16-bit width) instruction sets, as shown in Figure 2-1. Certainly there might be different instruction length besides 32 and 16 bits in a mixed width ISA. However there exists only these two instruction length in current mixed width ISAs. So the followed contents of our thesis are introduced in this presupposition.

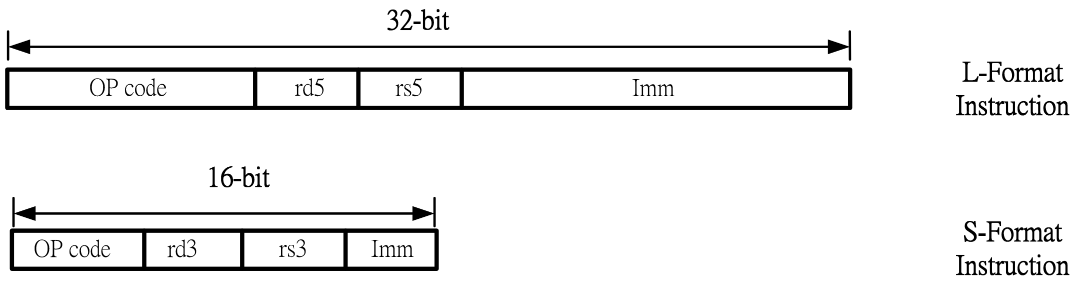


Figure 2-1: L- and S-Format instructions

In these ISAs, the program could be composed of L- and S-Format instructions and the code density may be improved. Because of the limitations of S-Format instructions, the usage of S-Format instructions could not be unlimited when compiling the program into native code. Moreover, there should be a mechanism to specify change between different instruction sets. There are two mode switch mechanisms used in present mixed-width ISAs. Subsequently, more details of limitations of S-Format instructions and mode switch mechanisms will be discussed in the following section.

2.1.1 Limitations of S-Format Instructions

In this section, limitations of S-Format instruction and how these limitations affect code size reduction will be described.

1. **Limited operations:**

   One of the limitations is that S-Format instruction set usually does not support all the operations include in L-Format instruction set. Since the number of bits of opcode is limited, only commonly used operations have S-Format instructions. Moreover, some complex operations need more fields and bits than S-Format instructions have. For example, in MIPS32/16, some L-Format instructions do not have equivalent S-Format, such as the instructions “abs.fmt”, “add.fmt”, “c.cond.fmt”, etc. Since these operations are few and seldom used, this limitation might only affect code size reduction slightly.

2. **Limited operand registers:**

The second limitation is that there are fewer bits for S-Format instructions to index operand registers. For example, in MIPS32/16, L-Format instructions have 5 bits to index operand registers, but S-Format ones have only 3 bits, i.e, there are only eight registers can be accessed by S-Format instructions.

In order to encode an instruction in S-Format, all of its register operands must be allocated to the S-Format indexable registers. Otherwise, if any one of its register operand is out of the register indexing range, this instruction must be encoded in L-Format.

Accordingly, the usage of registers has significant impact on S-Format translation rate. Unfortunately the traditional register allocation algorithms only try to allocate registers to reduce the amount of spill codes, but do not specify which registers should be allocated according to the register limitation of S-Format.

3. **Limited immediate value:**

The last limitation is that there are fewer bits for S-Format instructions to store immediate value. For example, in MIPS32/16, there are 16 bits for L-Format instructions to present immediate values but only 4~11 bits for S-Format ones. If the immediate value exceeds the range that can be presented by an S-Format instruction, then the instruction can only be encoded in L-Format.

Although large immediate values may have impact on the translation rate, the extent of this impact depends on how complier manages constants. If compilers use a constant pool to hold these large values, the affection can almost be neglected.

2.1.2 Mechanisms of Mode Switch

In this section, details of mechanisms of mode switch for mixed-width ISAs, such as how a program switches between two format instructions, are described. In the first mechanism, special instructions are used to switch a program between two format instructions. In the second one, instructions have specified encoding to indicate which format they are.

1. **Mechanism I: Switching by mode-switch instruction**

With first kind of mechanism, specific instructions, called “mode switch instructions”, are used while programs need to switch between L- and S-Format instructions. As shown in Figure 2-2(a), each function in the program consists of only L- or S-Format instructions with some necessary mode switch instructions when it calls or returns to different format function. For example, in ARM/Thumb, while switching between different format functions, the original “call“ and “return” instructions have to be replaced with proper mode switch instructions, such as “BLX”, “BL” etc. And then these mode switch instructions will switch program between two formats according to the information given by compiler. These extra mode switch instructions will cause code size increasing. Besides, some instructions lost opportunity to be formed in S-Format due to that all instructions in the same function usually are formed in the same format.

2. Mechanism II: Switching by instruction encoding

With the second kind of mechanism, each instruction needs to have a special encoding to notify processor which format it is. For example, as shown in Figure 2-3, use the MSB (most significant bit) of an instruction to indicate which format it is, where the MSB is “zero” or “one” means that the instruction is either L- or S-Format. Therefore, each instruction can be translate into S-Format individually if it meets the limitations of S-Format, as shown in Figure 2-2(b). In this case, there will be more opportunities to translate instructions into S-Format to reduce code size. Therefore, mixed-width ISA with mode-switch by instruction encoding becomes the trend in embedded RISC processor design.

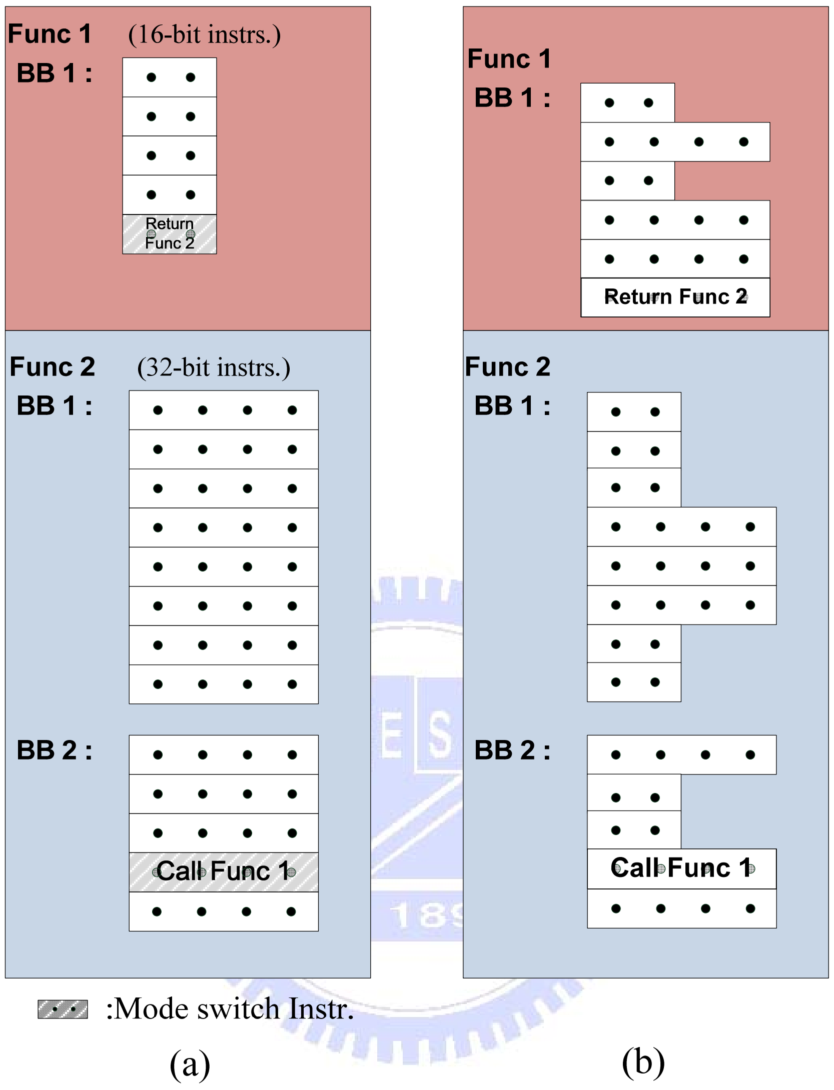


Figure 2-2: Example of a program which consists of L- and S-Format instructions with
(a) Mode switching by mode-switch instructions
(b) Mode switching by instruction encoding

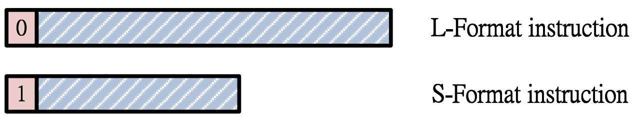


Figure 2-3: An example for mode switching by instruction encoding

2.1.3 Alignment Issue of Instruction Fetch

In a mixed width ISA processors, instruction alignment is an important issue which needs to be concerned. Usually the length of L-Format instructions is the twice of that of S-Format ones However, the instruction fetching width of processor is usually the same as the length of L-Format. Two different lengths of instructions in a program may cause misaligned problem.

This problem can be easily solved in mechanism I (switching by mode-switch instruction). For example, if there is odd number of instructions in a function with S-Format, misaligned problem occurs. In such case, an extra S-Format "nop" instruction needs to be added, as shown in Figure 2-4(a)

In mechanism II, the problem is harder to be solved because those L- and S-Format instructions are mixed in a program. To solve this problem the support of architecture is needed. While the processor fetches a word (32 bits) of instructions, one of three possible cases may be occurred, as shown in Figure 2-4(b). In case 1 and 2, one L-Format or two S-Format instructions are fetched in one cycle. So that processor can decode them without extra work. In case 3, the first half word is an S-Format instruction and the other half is the first part of another L-Format instruction. Therefore the other part of L-Format instruction is needed and provided by word fetched next cycle. So a fetch buffer is needed to store the first half of L-Format instruction temporarily. When the remained part of one L-Format instruction is acquired, the processor will combine two parts into one instruction and decode it. On the whole, instruction alignment problem in mixed width ISA is trivial but needed to be noticed.

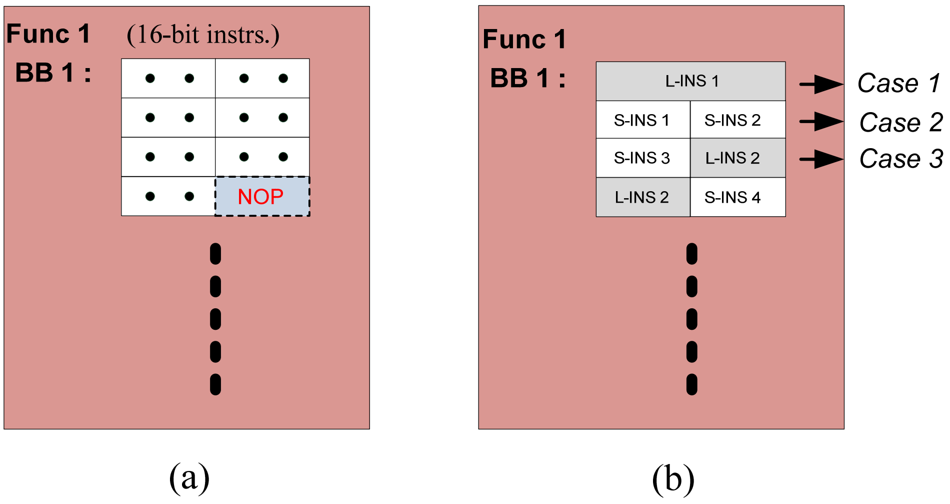


Figure 2-4: Example of misaligned problem in mixed-width ISAs
(a) Example for mode switching by mode-switch
(b) Example for mode switching by instruction encoding

2.2 Register Allocation Algorithms

Register allocation is an essential optimization in a compiler. While compiling a program, the compiler will assume that there are infinite temporal registers, called virtual registers, available at first. Then, the compiler will allocate physical registers to virtual registers in program before it emits native codes. Mapping these virtual registers to limited physical registers is a known NP-complete problem [8]. For this reason, the solution of nowadays register allocation algorithms is usually base on some known heuristics. There are two well known register allocation algorithms: Graph Coloring and Linear Scan register allocation.

In graph coloring register allocation algorithm, register allocation has been mapped to graph coloring problem to solve [8]. At, the algorithm analyzes the program and builds an interference graph. Each node in the graph presents a virtual register in program. If there is an edge exists between two nodes, it means that life times of these two virtual registers overlap. And then, the algorithm solves register allocation as solving graph coloring problem. This algorithm has variant versions and been widely used in compilers in the present day. The time complexity of the algorithm is $O(n^2)$, where n is the number of virtual registers. This algorithm usually produces efficient results in a reasonable time.

In linear scan register allocation algorithm (LSRA), all virtual registers are treated as intervals which present the flat life time of virtual registers and allocated to physical registers in a linear scan [9]. In the beginning, the algorithm analyses the program and builds an interval list. The intervals are stored in this list and sorted in order of increasing start points. And then, the algorithm scans the list and allocates registers to each interval in linear time. This algorithm usually produces less efficient code than graph coloring algorithm does, but its time complexity is O(n).

In dynamic compilers, such as just-in-time compiler in JAVA virtual machine, compilation time is an important issue. So, optimizations in it must be fast and efficient. In consideration of this demand, linear scan register allocation is suitable to use in dynamic compilers. Because of this reason, the algorithm which we proposed is designed based on linear scan register allocation. Detail of linear scan algorithm will be described in the following subsections.

2.2.1 Live Intervals

LSRA treats virtual registers as intervals while allocating physical registers. These intervals called "live intervals" which present how long virtual registers live in program. While building intervals, the liveness of each virtual register is needed. This information will be calculated by an iterative pass called liveness analysis [10]. The pass will calculate correct liveness for each virtual registers by analysis control flow graph. Before introducing LSRA there are some structures and states needed to be known first.

As shown in Figure 2-5, start and end points of intervals are the positions where the first instruction defines the value of the virtual register and the last instruction use it. Meanwhile defined point is the positions where the instructions define the value of the virtual register. Use point is the positions where the instructions use the value of the virtual register. Live intervals can be built by one pass program scan. LSRA only needs to know start and end points of intervals while allocating registers, but our design also needs to know the define and use points .

There are three possible states of intervals while LSRA processing intervals. One is unhandled state; it means that the interval has not been processed by LSRA. State of all intervals is unhandled initially. Another is unexpired state; it means that the interval has been processed and still live in this time point. The other is expired state; it means that the interval is no longer live.

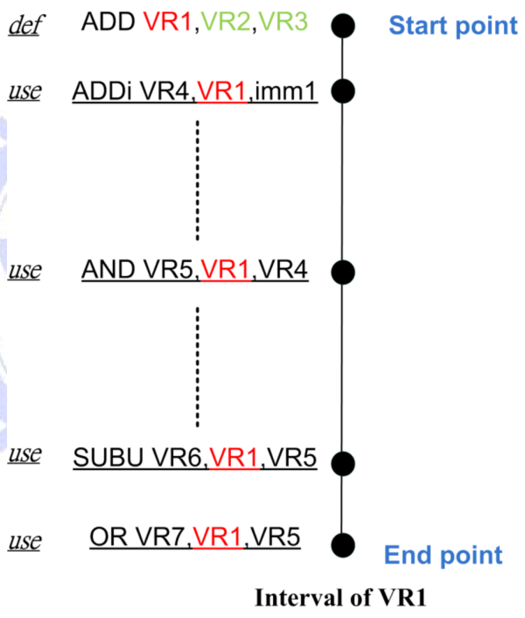


Figure 2-5: Start, end and use point at an interval

2.2.2 Algorithm of LSRA

The overall flowchart of linear scan register allocation algorithm is shown in Figure 2-6.

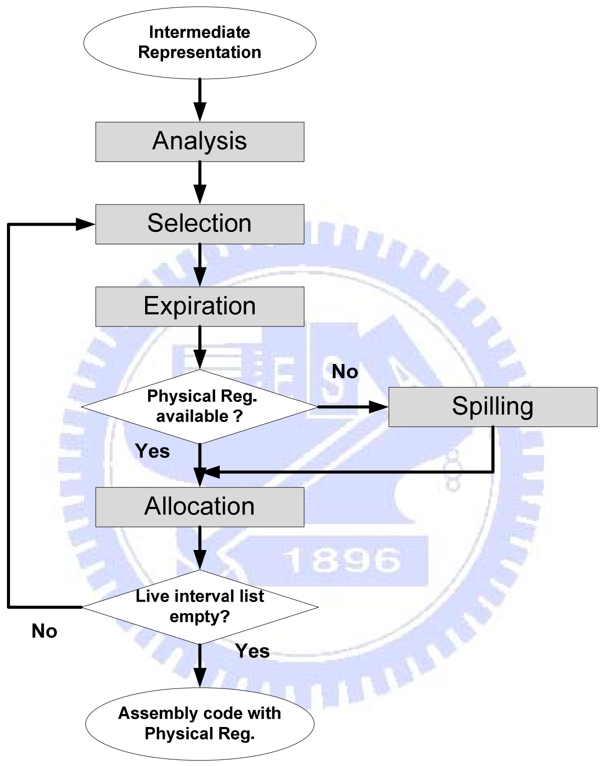

Intermediate Representation
Analysis
Selection
Expiration
Physical Reg. available ?
No
Spilling
Yes
Allocation
Live interval list empty?
No
Yes
Assembly code with Physical Reg.

Figure 2-6: Flow chart of linear scan register allocation

First of all, the algorithm analyzes the program to build an intervals list (LI) and prepares an empty register active list (RAL). LI stores the intervals which have not been processed and stores them by the increasing order of their start points. RAL stores the unexpired intervals by thedecreasing order of their end points and records which physical registers are occupied.

Then, LSRA selects an interval whose start point is the smallest of all from the LI, and uses the start point of the selected interval to remove those intervals that end before it from RAL. After that, LSRA finds a free register and allocate it to the selected interval.

If there is no free register, the algorithm will choose an interval whose end point is farthest from RAL and compare it to the newly selected interval. If the end point of the interval from RAL is farther than the new interval, LSRA will spill the interval from RAL and free the occupied physical register, and then allocate this register to the newly selected interval. Otherwise, LSRA spills the newly selected interval. After scanning all intervals, the compiler maps all virtual register to physical registers and generates necessary spill codes.

2.3 Summary of Background

For mixed-width ISA with mode-switch by instruction encoding, we have study the effects caused by the limitations of S-Format on code size reduction. As describe in section 1.1 and 2.1, we found that the limitation of operand registers is the most critical one. This is because that traditional register allocation algorithm assigns register number casually. The reason why traditional register allocation algorithms do that is because they consider the number of spill code only but not which physical register should be assigned to each virtual register, and this may cause that less instructions can be translated into S-Format. Therefore, the traditional register allocation does not suit for this problem apparently. For this reason, in order to increase translation rate of S-Format instruction, we have to design a specific register allocation algorithm. Moreover, we will design this algorithm based on LSRA because we want to apply it on JIT compiler.

Chapter 3
Hierarchical Linear Scan Register Allocation

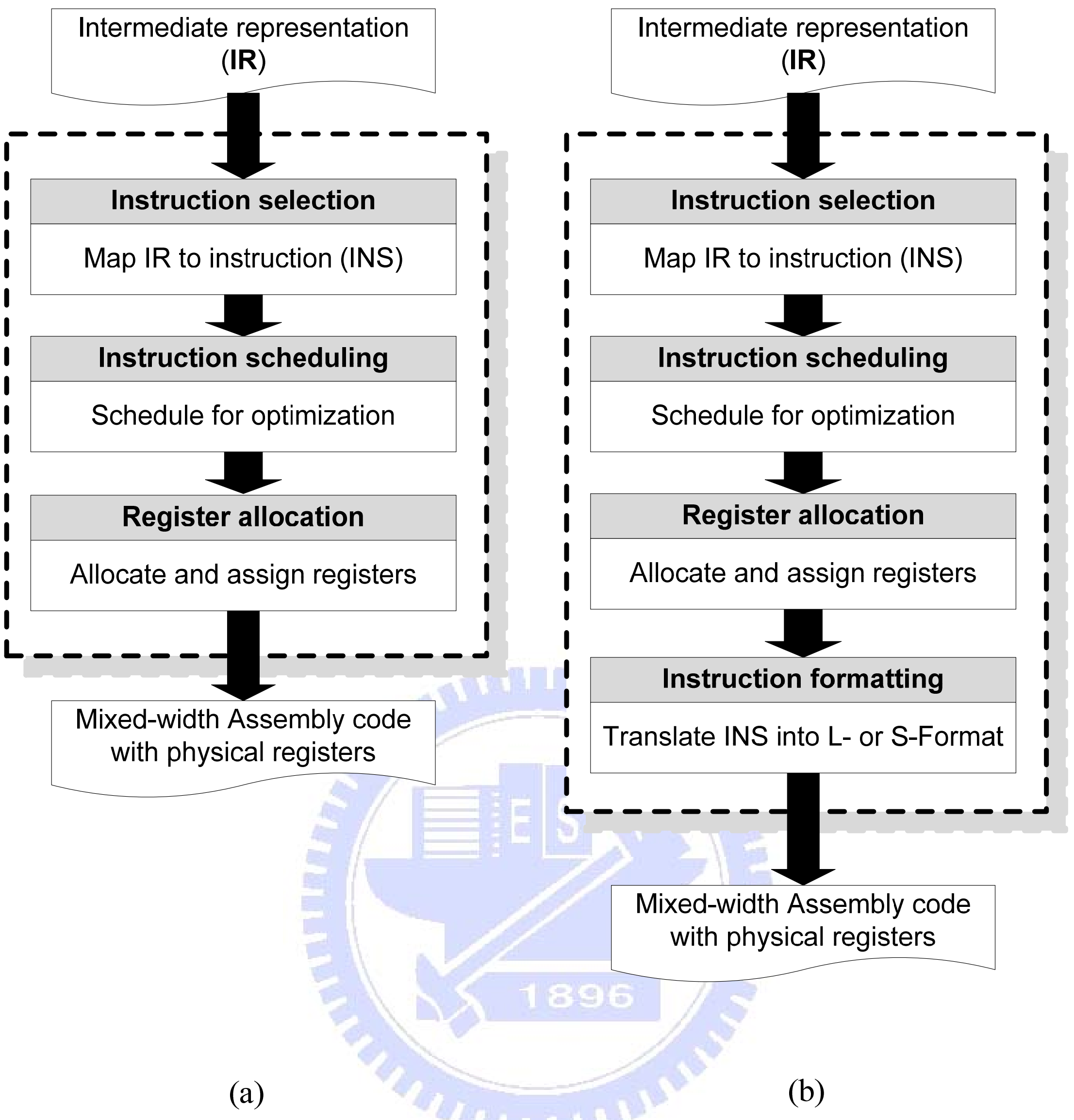
This thesis proposes a fast register allocation algorithm that not only determines which virtual register should be stored in the physical register to decrease spill code, but also evaluate which virtual register should be assigned to the register can be indexed by S-Format instruction. This specific algorithm is based on linear scan register allocation (LSRA) and called "Hierarchical linear scan register allocation (HLSRA)". We choose LSRA as foundation is because of its fast compilation time and efficient. Therefore it is suitable for dynamic compiler such as just-in-time compiler in JAVA virtual machine.

The basic idea of the proposed algorithm is allocating the registers which can be accessed by S-Format instruction to the virtual registers that are most commonly used.

3.1 Compiler Back-end for Mixed-width ISA

In general, traditional compiler back-ends consist of three stages, which are instruction selection, instruction scheduling, and register allocation, as shown in Figure 3-1(a). The first stage, instruction selection, maps intermediate representations (IR) to correspondent machine instructions, where the operand registers of these instructions are virtual registers. The second stage, instruction scheduling, schedules instructions for hiding possible pipeline stall to increase performance. Finally, the third stage, register allocation allocates physical registers to virtual registers.

However, in mixed-width ISA with mode-switch by instruction encoding, an instruction may be represented in L- or S-Formats. Therefore, an extra stage is added into compiler back-end for this kind of ISAs, as shown in Figure 3-1(b). In this compiler back-end, the first stage is instruction selection, where this stage maps an operation to temporary instruction called INS. Each INS may have L- or both L- and S-Format. The second stage, instruction scheduling does the same thing as in traditional compiler back-end. The third stage, register allocation may apply specific algorithm for mixed-width ISA or not while allocating registers. At the end, the added stage, instruction formatting, decides which format should instructions be according to their opcode and operands (e.g. register operands and immediate value). For example, if an arithmetic INS "add" has L-Format and S-Format instructions as shown in Figure 3-2, the instruction formatting stage will check whether this INS meets all limitations of S-Format instruction. If it does, then instruction formatting translates "add" into "adds"; if not, it will be formed into L-Format. Finally assembler will encode the instructions in corresponding formats.



Intermediate representation (IR)
Instruction selection
Map IR to instruction (INS)
Instruction scheduling
Schedule for optimization
Register allocation
Allocate and assign registers
Mixed-width Assembly code with physical registers
(a)
Intermediate representation (IR)
Instruction selection
Map IR to instruction (INS)
Instruction scheduling
Schedule for optimization
Register allocation
Allocate and assign registers
Instruction formatting
Translate INS into L- or S-Format
Mixed-width Assembly code with physical registers
(b)

Figure 3-1: Different frameworks of Compiler back-end

(a) Traditional compiler back-end

(b) Compiler back-end for mixed-width ISA

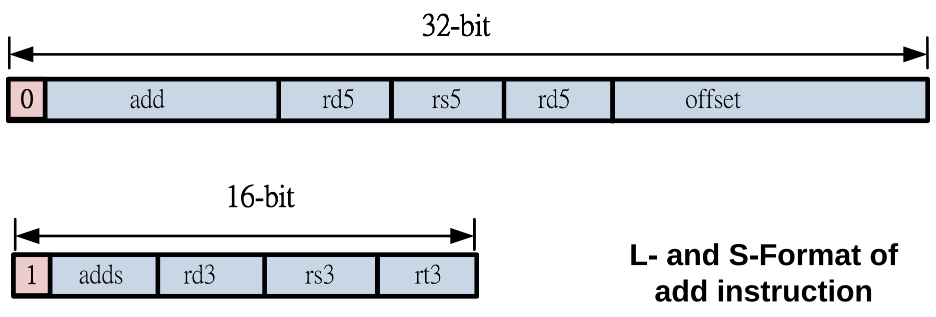


Figure 3-2: L- and S-Format of arithmetic INS

3.2 Design of HLSRA

In this section we will define some terms and structures for our design. Firstly we classify instructions of program into three types. And then we describe the data structures (e.g., register set, register active list) and priority functions which is needed in our design. At last we will describe the algorithm of our design

3.2.1 Instruction Types

In mixed-width ISA, most instructions have multiple formats. For our design, we classify instructions in program into three types.

1. **L-INS (L-Format INS)**: L-INS is an instruction without S-Format or which cannot be encoded in S-Format because it has exceeding immediate value.

2. **S-INS (S-Format INS)**: S-INS is an instruction which has both L- and S-Format, and definitely meets all limitations of S-Format, for examples, JR and NOP. JR has only one operand register so that it has enough bits for indexing register, so it can access the whole physical registers. Furthermore, NOP does not have any operands, thus, it can be encoded in S-Format definitely.

3. **U-INS (Uncertain-Format INS)**: U-INS is an instruction which has both L- and S-Format and can be translated into S-Format if and only if it meets limitation of operand registers.

3.2.2 Register Sets

In our design, the physical registers are divided into two sets, RegisterSets_S and RegisterSet_L. S-Format instruction can only access the registers in RegisterSets_S, but L-Format instruction can access the registers in both sets. As shown in Figure 3-3, for example, if there are three bits in S-Format instruction and five bits in L-Format for register indexing, then register r0~r7 are in RegisterSets_S and register r8~r31 are in RegisterSet_L. Register_S and Register_L represent the registers in RegisterSets_S and RegisterSet_L respectively. The number of free registers in RegisterSets_S and RegisterSet_L are named as FRSN_S and FRSN_L.

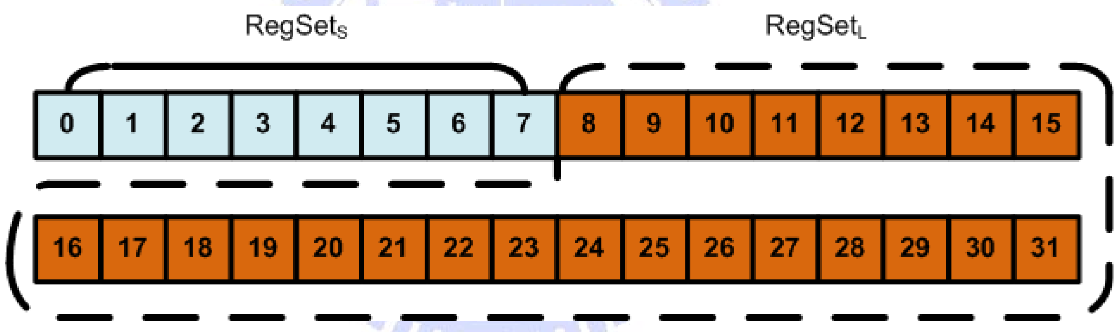


Figure 3-3: RegisteSets_S and RegisterSet_L

3.2.3 Register Active Lists of HLSRA

In traditional LSRA, there is only one register active list (RAL) and its size is the number of physical registers. Because HLSRA divides original register set into two separate register sets, it has to prepare two register active lists, called RAL_S and RAL_L. The size of RAL_S and RAL_L are the same as the size of RegisterSets_S and RegisterSet_L. Besides, HLSRA has to maintain these RALs and record the interval which has minimal priority during register allocation. The intervals which have minimal priority in RAL_S and RAL_L are named as mI_S and mI_L.

3.2.4 Priorities of HLSRA

While allocating registers to intervals, if there is no free physical register, an interval need to be spilled. In LSRA, whether an interval should be spilled or not depends on its end-point. In HLSRA, we use a different heuristic approach to decide which interval should be spilled.

In our design, the physical register set is divided into two sets, $RegisterSet_S$ and $RegisterSet_L$. Therefore there are two types of priorities, $Priority_S$ and $Priority_L$, for allocating registers in these register sets respectively. When there is no free register in these register sets, priorities are used to decide which interval should be spilled. For example, while HLSRA trying to allocate register in $RegisterSet_S$ and there is no free register, it will use $Priority_S$ to choose spilled interval. These priorities are estimated according to the information about intervals. Since $Priority_S$ and $Priority_L$ are used to handle different situation, they take different parameters into consideration.

$Priority_S$ take three things into consideration. Firstly HLSRA want to make as many U-INSs as possible to meet limitation of operand registers, thus it takes the number of U-INSs which use the value of the interval into consideration. In the second, if the instruction which defines the interval is U-INS, we will also consider the number of operand registers which has been assign to $Register_S$ of this instruction. For example, if there are two $Register_S$ used for operands of the U-INS, this U-INS will be translated into S-Format while a $Register_S$ is allocated to the interval which is defined by the U-INS. At last, we account life range of the interval too. Life range presents the time that an interval will occupy a register. If a physical register is allocated to the interval whose life range is short, the register won't be occupied too long. Therefore this register will be available soon and allocated to other intervals. Thus, the interval whose life range is shorter than others should have higher priority to acquired physical register. The priority function of Priority_S is as follow

$$\text{Priority}_S = \frac{\alpha \times N_1 + (1-\alpha) \times S}{LR} \quad \ldots\ldots\ldots (1)$$

N_1 presents the number of U-INSs which use the value of the interval; S presents the number of operand registers which is Register_S of the instruction which first defines the interval. LR presents life range of the interval. For example, as show in Figure 3-4, the number of U-INSs which use interval(VR1) is $N_1 = 3$, and the number of allocated operand register which is Register_S is $S = 2$, and life range of the interval is $LR = 8$.

Priority_L is similar to Priority_S. First Priority_L consider the number of instructions which use the interval; the more instructions use the value of the interval, the more spill codes will be produced if it is spilled. Second, life range of the interval is also taken into consideration like Priority_S. The priority function of Priority_L is as follow

$$\text{Priority}_L = \frac{N_2}{LR} \quad \ldots\ldots\ldots (2)$$

N_2 presents the number of instructions which use the interval and LR presents life range of the interval. For examples, as show in Figure 3-4, the number of instructions which use interval(VR1) is $N_2 = 4$ and life range of the interval is $LR = 8$.

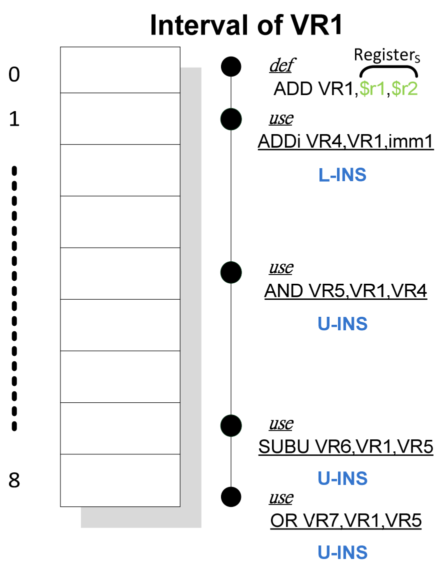


Figure 3-4Figure 3-4: An example for priorities of intervals

.

3.2.5 *Spilling of HLSRA*

In LSRA, the interval which has not been allocated to any physical register is spilled into memory directly. In our design, the interval which has not been allocated to Register$_S$ might be allocated to Register$_L$. Although those U-INSs which use this interval cannot be translated into S-Format due to the register limitation, allocating Register$_L$ to the interval might bring some benefit (such as less spill code). For this reason, there are Spill$_S$ and Spill$_L$ for storing the interval which will be spilled. The intervals in Spill$_S$ and Spill$_L$ are named as I$_S$ and I$_L$.

Spill$_S$ stores the interval which has not been allocated to Register$_S$. Subsequently, HLSRA will remove the interval from Spill$_S$ and try to allocate it Register$_L$ rather than spill it into memory directly.

Spill_L stores the interval which has not been allocated to Register_L. According to our design, this interval has no chance to get physical register so spilling pass will remove the interval from Spill_L and spill it into memory.

3.2.6 *Algorithm*

The basic idea of our design is allocating registers in RegisterSet_S and RegisterSet_L sequentially. As show in Figure 3-5, for example, while allocating a register to an interval, such as interval (VR_{56}), HLSRA allocates Register_S first. If there is no free register in RegisterSet_S, HLSRA will compare Priority_S of interval (VR_{56}) and the interval which has lowest Priority_S in RAL_S (mI_S). According comparison result, the interval which has higher Priority_S will be allocated to Register_S. Next, HLSRA will try to allocate Register_L to the interval which cannot be allocated to Register_S. If there is still no free registers, HLSRA will select an interval in RAL_L with lowest priority and spill it into memory.

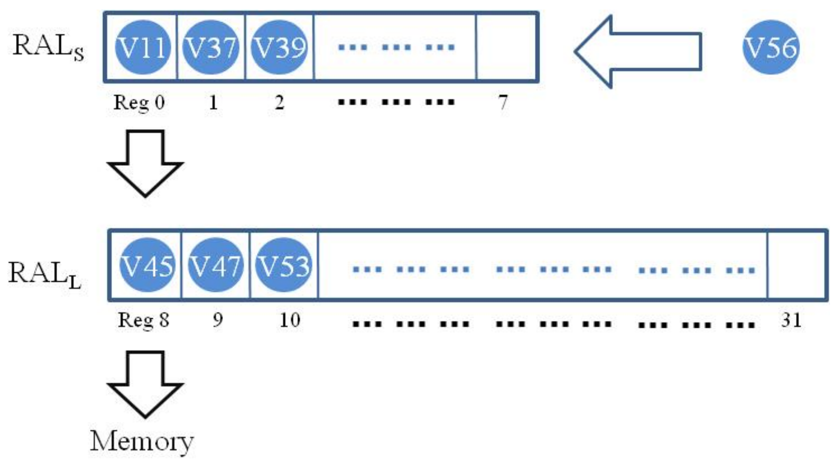


Figure 3-5Figure 3-5: An example of HLSRA

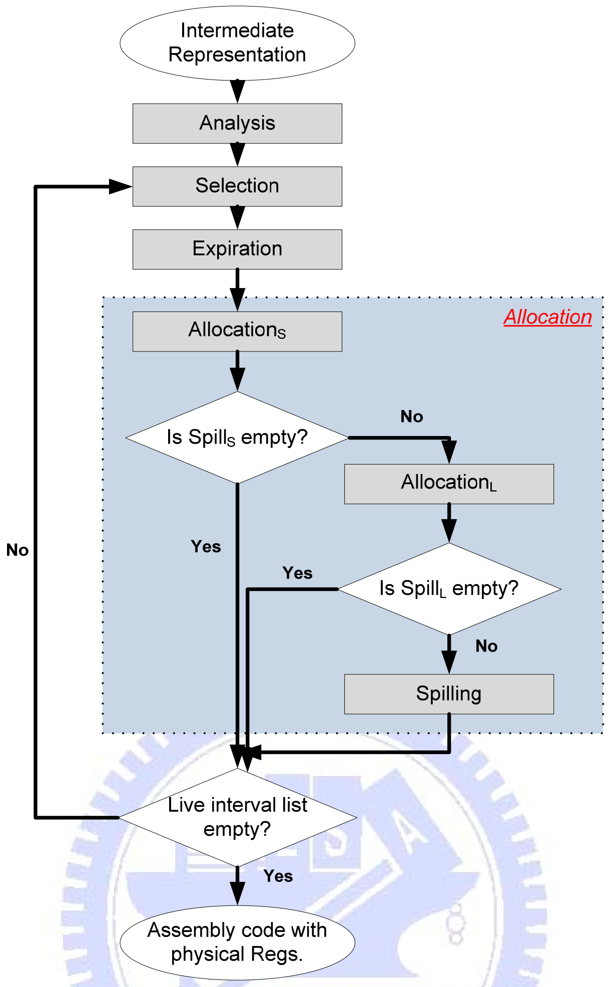


Figure 3-6Figure 3-6: Hierarchical Linear Scan Register Allocation

The overall flow chart of the HLSRA is as shown in Figure 3-6, HLSRA allocates physical registers hierarchically. At first, analysis pass scans the program to build live intervals, gets interval information needed by HLSRA (such as the number of U-INSs which use this interval, life range, etc) and stores these intervals by increasing order of their start point into interval list (LI). Then selection pass selects an interval from LI, and then expiration pass uses the start point of selected interval to expire the intervals in RAL$_S$ and RAL$_L$. After expiration pass, HLSRA starts allocating a physical register to the selected interval.

Allocation$_S$ processes the selected interval and tries to allocate Register$_S$. In the beginning Allocation$_S$ checks FRSN$_S$ to see if there are available registers in RegisterSet$_S$. If FRSN$_S$ is greater than zero, Allocation$_S$ will allocate one of these free registers to the selected interval. While FRSN$_S$ is equal to zero, there is no free register available. In this case, Allocation$_S$ has to decide which interval should be placed into Spill$_S$. The interval in Spill$_S$ will be processed by Allocation$_L$ later.

To determine which interval should be put into Spill$_S$, Allocation$_S$ chooses the interval whose Priority$_S$ is smallest from RAL$_S$ called mI$_S$, and compares it with the selected interval. If Priority$_S$ of selected interval is greater than Priority$_S$ of mI$_S$, Allocation$_S$ will remove mI$_S$ from RAL$_S$ and put it into Spill$_{S,}$ and then the Register$_S$ which was occupied by mI$_S$ will be allocated to the selected interval; otherwise the selected interval will be put into Spill$_S$.

If Spill$_S$ is not empty after finishing Allocation$_S$, Allocation$_L$ will remove the interval called I$_S$ from Spill$_S$ and process it; otherwise HLSRA finish this round of iteration.

To processing I$_S$, like Allocation$_S$, Allocation$_L$ checks FRSN$_L$ to see if there are available registers in RegisterSet$_L$. If FRSN$_L$ is greater than zero, Allocation$_L$ will allocate one of these free registers to I$_S$. If FRSN$_L$ is equal to zero, there is no free register available. In this case, Allocation$_L$ selects the interval whose Priority$_L$ is smallest from RAL$_L$ called mI$_L$, and compares it with the interval (I$_L$) of Spill$_S$. If Priority$_L$ of I$_L$ is greater than Priority$_L$ of mI$_L$, Allocation$_L$ will remove mI$_L$ from RAL$_L$ and put it into Spill$_L$, and then the Register$_L$ which was occupied by mI$_L$ will be allocated to I$_S$; otherwise put I$_S$ into Spill$_L$.

If Spill$_L$ is not empty after finishing Allocation$_L$, spilling pass will be started to spill the interval in Spill$_L$; otherwise HLSRA finish this round of iteration. The pseudo code of HLSRA is shown in Figure 3-7.

```
Procedure HLSRA
// intervals stored in LI are sorted by increasing order of start point
forall interval$_i$ ∈ LI do
    Calculate priorities for interval$_i$
    Expire($RAL_S$) and Expire($RAL_L$)
    if $FRSN_S > 0$ then                                              ┐
        Allocate $Register_S$ to $interval_i$                           │
    else                                                              │
        Compare Priority$_S$($mI_S$) to Prriority$_S$($i$)                  │
        if Priority$_S$($mIs$) < Priority$_S$($i$) then                     │
            Remove $mI_S$ and put it into $Spill_S$                     │  Allocation$_S$
            Allocate $Register_S$ which is occupied to $interval_i$     │
            Update RAL$_S$                                            │
        else                                                          │
            Put $interval_i$ into $Spil_S$                              │
        endif                                                         │
    endif                                                             ┘
    if $Spill_S$ ≠ ∅ then
        if $RSN_L > 0$ then                                           ┐
            Allocate $Register_L$ to $I_S$                              │
        else                                                          │
        Compare Priority$_L$($mI_L$) to Prriority$_L$($I_S$)                │
            if Priority$_L$($mI_L$) < Priority$_L$($I_S$) then              │
                Remove $mI_L$ and put it into $Spill_L$                 │  Allocation$_L$
                Allocate $Register_L$ which is occupied to $I_S$        │
                Update $RAL_L$                                        │
            else                                                      │
                Put $I_S$ into $Spill_L$                                │
            endif                                                     │
        endif                                                         ┘
    if $Spill_L$ ≠ ∅ then
        Spill  $I_L$
    endif
endfor
```

Figure 3-7Figure 3-7: Algorithm of HLSRA

Chapter 4 Experiment

In the first part of this chapter, we will describe the experiment environment, including the instruction set architecture, used compiler back-end and benchmarks. In the second part, we will show how to decide parameters using in our experiments. At last, we will discuss the evaluation result about S-Format translation rate and influence of spill code for HLSRA.

4.1 Environment

In our experiments, we propose a pseudo mixed-width ISA with mode switch by instruction base on MIPS32/16, called MIPSe. All of the instructions have same number of formats in MIPSe and MIPS32/16, for example, instruction "add" has L- and S-Format in MIPSe as same in MIPS32/16. There is only switch mode of MIPSe different from MIPS32/16. In MIPSe, we assume that MSB of instruction op-code indicates the format of this instruction. All benchmarks used in this experiment were selected from SPEC INT2000 [11], Mibench[12], and Mediabench[13].

According to MIPS32/16 ISA and our design, the available registers in each register sets are as follows: RegisterSet_S = r1 ~ r7 (r0 is used as zero in MIPS32/16, so it cannot be used) and the number of $\text{FRSN}_S = 7$; RegisterSet_L = r10~r25 (r8~r9 is reservedto process spill code, r26~r27 is reserved for OS kernel and r28~r31 are reserved for calling convention) and the number of $\text{FRSN}_L = 16$.

The Low-Level Virtual Machine (LLVM) [14] is a compiler infrastructure. It provides language- and target-independent components for building compilers. There are several targets which are supported by LLVM, such as MIPS, ARM, x86, PowerPC, etc. We modified the LLVM compiler back-end to produce the INSs with virtual registers as inputs for HLSRA.

4.2 Parameter Determination

While calculating Priority_S for each interval, HLSRA have to concern about the number of U-INSs that use the value of the interval (N_1) and the number of operand registers which are Register_S of the instruction which first defines the interval (S). In another words, we have to adjust α which controls the weight between N_1 and S to get best S-Format translation rate. In our experiments, S-Format translation rate means that the percentage of the instructions which can be translated into S-Format over all instructions. The results are shown in Figure 4-1.

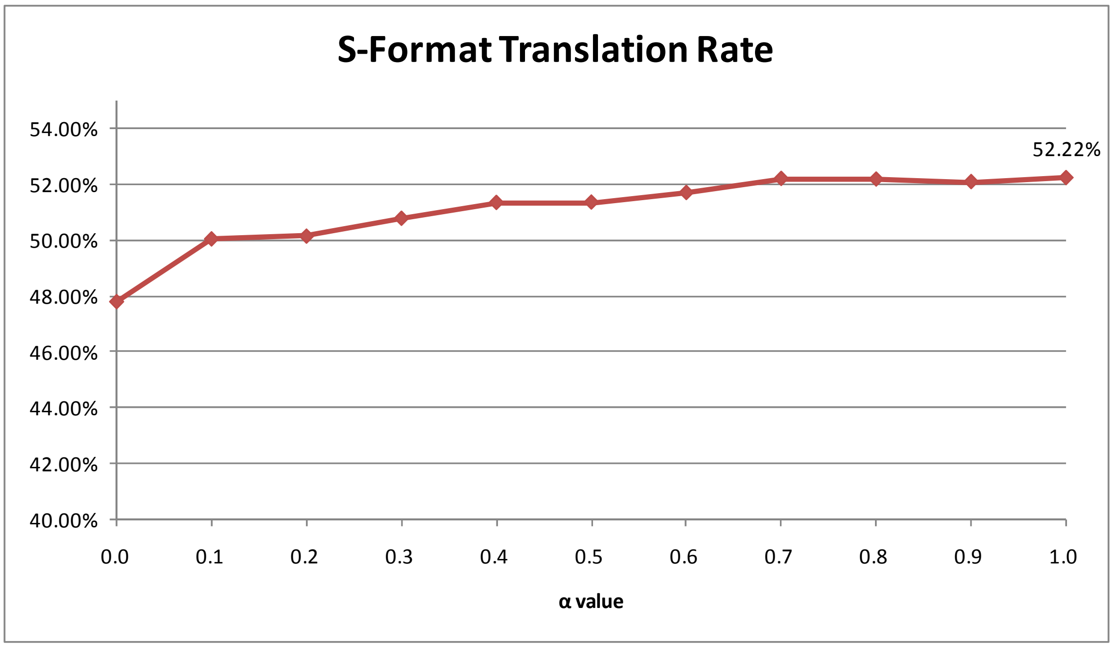


Figure 4-1Figure 4-1: S-Format translation rate

In the above figure, X-axis presents α value, Y-axis presents S-Format translation rate. HLSRA can achieves best S-Format translation rate when α equals one. Obviously, the number of U-INSs that use the value of the interval is the most important parameter for HLSRA to reduce code size.

4.3 Evaluation Results

In this section, we will illustrate evaluation results for HLSRA. The value of α is set to 1.0 according to the result obtained in section 4.2. As shown in Figure 4-2, we illustrate evaluation result for HLSRA and compare it to the result of LSRA. As expected, HLSRA achieves 52.22% S-Format translation rate and there is 13% more than which LSRA can make.

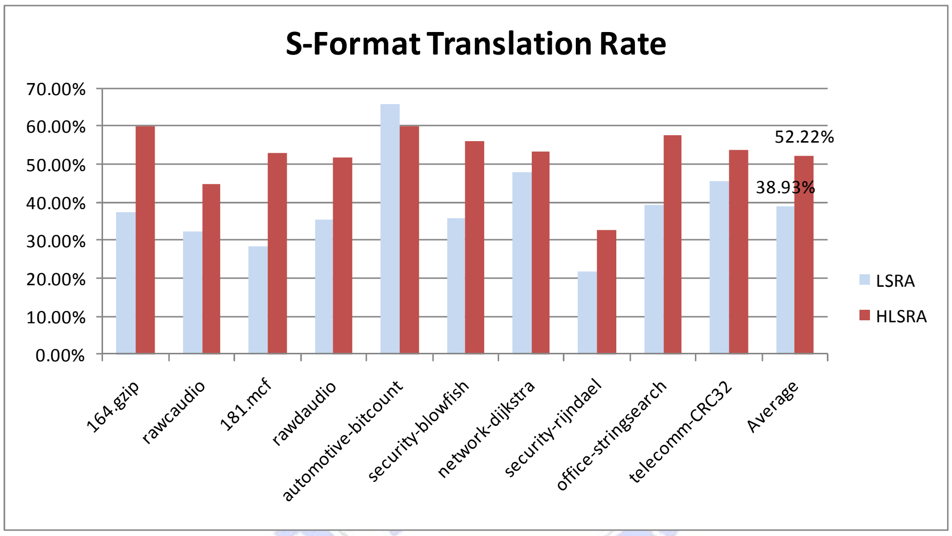


Figure 4-2: S-Format translation rate

As shown in Figure 4-3, number of spill codes which produced by HLSRA accounted for 8% of all instructions on the average. We can see that HLSRA has generated slightly more spill code than LSRA. The reason why this happens is described below. LSRA decides which interval should be spilled according to their end point to decrease number of spill codes. Although how HLSRA decides which interval should be spilled is similar to LSRA, HLSRA chooses interval according to their priorities to decrease code size. Therefore, HLSRA will produce more spill codes than LSRA

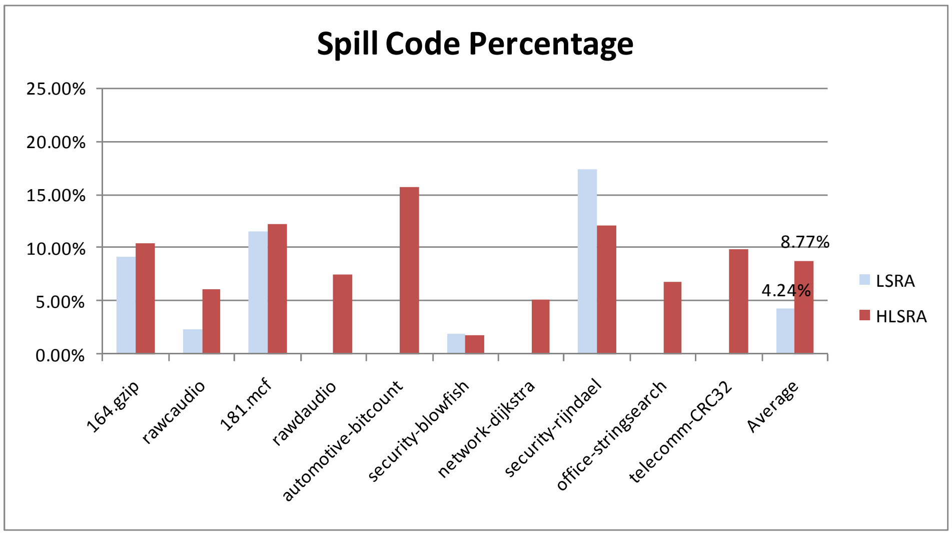


Figure 4-3: Spill code percentage

In Figure 4-4, we show the code size reduction achieved by HLSRA and compare it to the result of LSRA. The code size reduction is the number of instructions which can be translated into S-Format divide by two since each L-format instruction is twice the length of an S-format instruction.

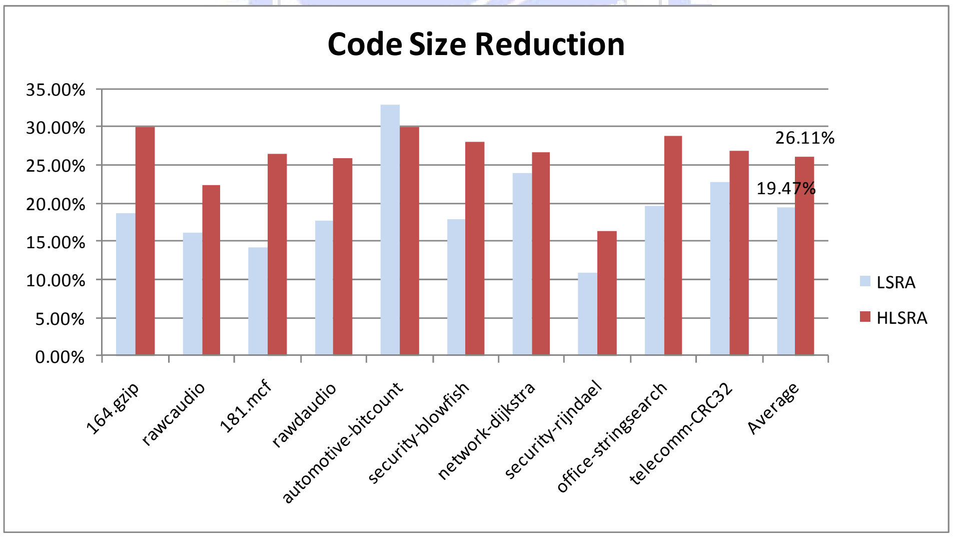


Figure 4-4: Code size reduction

At last, we illustrate distribution of S-Format limitations in all programs which were produced by HLSRA and compare it to LSRA, as shown in Figure 4-5. The Y-axis is the percentage of instructions. In the figure, the top block, green colored, is the percentage of instructions which have no S-Format instructions or the immediate value is oversized. The middle block, red colored, is the percentage of instructions which cannot be formed in S-Format due to register limitation. And the bottom block, blue colored, is those instructions whose register number and immediate value are in the range of S-Format instructions.

The number of instruction which can be translated into S-Format is increased in programs processed by HLSRA. The HLSRA eases the problem of register limitation and makes 14% more instructions converted into S-Format than LSRA does. These instructions were not formed in S-Format due to register limitation originally in code produced by LSRA. By the proposed HLSRA, about 52% translation rate of S-Format instructions in programs is achieved

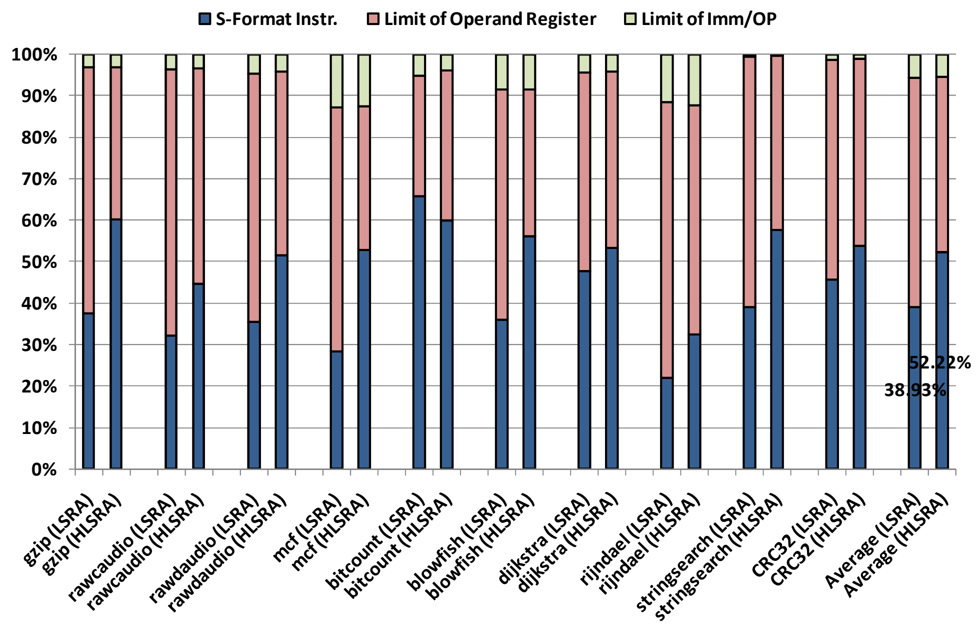


Figure 4-5: S-Format limitation distribution

Chapter 5 Conclusions and Future Work

In this chapter, we will make conclusions and discuss some possible future works for this thesis.

5.1 Conclusion

In this thesis, we presented hierarchical linear scan register allocation (HLSRA) algorithm which allocates registers more friendly for mixed-width ISA with mode switch by instruction encoding than traditional register allocation algorithms like LSRA. Because the registers which can be accessed by S-Format instruction are allocated to intervals which have been mostly used by U-INSs, HLSRA is supposed to ease the affect brought by limitation of operand registers for code size reduction.

To achieve the goal described previously, we modify the allocation pass to fit the demand of mixed-width ISA, where allocation pass has been separated into two parts. Compared to LSRA, this modification will not increase time complexity. In addition, this design only brought neglectable overheads, such as producing a little more spill codes than LSRA and extra data structure, such as Spill_S and Spill_L.

Overall, our design makes S-Format instruction rate increasing to about 52% on the average. As discovered in Section 4.3, there are still 42% instructions could not be formed in S-Format due to register limitation in HLSRA. The reason why this part cannot be improved by our design is described as follow. Firstly, because there exist over eight intervals live simultaneously at many execution points in the programs, there are certainly some instructions which cannot be formed in S-Format due to register limitation. Secondly, HLSRA cannot use more efficient but complex methods, such as iterative heuristic policies, because of the demand of low compilation time for dynamic compilers.

5.2 Future Work

As we mention in section 5.1, since the design for dynamic compiler must be fast, complex optimizations are not suitable. Consequently, there are some aspects of future works for our thesis.

[1] The First possible future work is to modify HLSRA to get better S-Format translation rate. Perhaps we can modify our design with a little more time consuming policy. There are some possible modifications.

The First possible modification is adapting a little more complex priority calculation methods, for example, building interference graph for small scope of intervals and using this graph to calculate priorities.

The Second one is to split interval while spilling it. Our design will spill whole interval when it cannot get a register rather than split it. When HLSRA spills an interval, it can insert extra store instructions into this interval to split it into two intervals: previous and remain intervals, and spills remain interval only. In this way, the previous interval will not be spilled and less spill code will be produced.

[2] The second possible future work is to apply other fast optimization for increasing S-Format translation rate. As introduced in Section 3.1, there are instruction selection, instruction scheduling and register allocation in compiler-backend. In our thesis, a specific register allocation is proposed for mixed-width ISA to increasing S-Format translation rate. Therefore using a specific instruction scheduling for mixed-width ISA may achieve the same goal.

The purpose of traditional instruction scheduling is decreasing instruction dependency for performance. For increasing S-Format translation rate, the specific instruction scheduling could consider the more information while scheduling instruction, such as which formats instructions have, etc.

[3] The third possible future work is applying our design to some other commercial mixed-width ISAs with mode switch by instruction encoding, such as ARM's Thumb-2, MIPS' new microMIPS. We can get more information about how our design affects code size reduction and execution performance through this work. Presently we cannot do that because there are no suitable compiler backends for these ISAs.

Reference

[1] Aviral Shrivastava, Partha Biswas, Ashok Halambi, Nikil Dutt, Alex Nicolau, "Compilation Framework for Code Size Reduction Using Reduced Bit-Width ISAs (rISAs)", ACM Transactions on Design Automation of Electronic Systems, Vol. 11, No. 1, January 2006, Pages 123–146.

[2] Arvind Krishnaswamy, Rajiv Gupta, "Mixed-Width Instruction Set", Communications of the ACM, v.46, n.8, August 2003.

[3] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, *Java Language Specification*, Addison-Wesley Longman Publishing, 2000.

[4] Tim Lindholm, Frank Yellin, *Java Virtual Machine Specification*, Addison-Wesley Longman Publishing, 1999.

[5] ARM Architecture Reference Manual, http://www.arm.com/

[6] MIPS32™ Architecture For Programmers, http://www.mips.com.tw/

[7] AndeStar™ ISA, http://www.andestech.com/

[8] Gregory Chaitin, "Register allocation and spilling via graph coloring", Computer Languages, Vol. 6, pp. 47-57, 1981.

[9] Massimiliano Poletto, Vivek Sarkar, "Linear scan register allocation", ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 21 , Issue 5, (September 1999) Pages: 895 – 913.

[10] Andrew W. Appel, Jens Palsberg, *Modern Compiler Implementation in Java,* Cambridge University Press, 2002

[11] SPEC INT2000 http://www.spec.org/cpu2000/CINT2000/

[12] Mibench http://www.eecs.umich.edu/mibench/

[13] Mediabench http://euler.slu.edu/~fritts/mediabench/

[14] Low-Level Virtual Machine, http://llvm.org/