Code Compaction for VLIW Instructions

by

Kai-Ting Amy Wang

A thesis submitted in conformity with the requirements
for the Degree of Master of Applied Science in the
Department of Electrical and Computer Engineering
University of Toronto

Code Compaction for VLIW Instructions

Kai-Ting Amy Wang
Master of Applied Science, 2001
Edward S. Rogers Sr.
Department of Electrical and Computer Engineering
University of Toronto

Abstract

The VLIW architecture is modular and scalable which makes it suitable for implementing application-specific programmable processors. However, the large instruction storage and bandwidth requirements have prevented VLIW architectures from being used in cost-sensitive systems.

This thesis builds upon a novel instruction packing and fetching mechanism developed by previous University of Toronto researchers, by exploring ways to achieve code compaction via improved packing techniques, stack restructuring and register allocation. The improved packing techniques involve transforming an instruction into a more compact-able equivalent instruction and are able to achieve 100% memory utilization and an average of 2.4% size reduction. Stack restructuring focuses on minimizing the unique stack-related constant operands, which in turn minimizes the number of unique operation patterns, leading to a maximum of 16.8% memory size reduction. Finally, live-range recoloring techniques are developed, targeting on minimizing the total unique operation patterns. Unfortunately, there are no significant reductions achieved.

Two of the three aspects examined in the thesis are able to further reduce the instruction storage requirement without affecting the execution speed of the program.

Acknowledgements

I am extremely grateful to the many individuals who contributed towards the success of this thesis. First and foremost, I want to express my deepest thanks to Professor Paul Chow for his patience with my never-ending questions, his support in all my technical hindrances and his always positive encouragement. Without him, this study would not have been possible and I am thankful for the opportunity to learn from him and benefit from his insights as a professor, mentor and friend.

Tor Aamodt is another individual whose assistance has been integral to the success of this project. Even though Tor is always very busy, he is very willing to spare time to discuss the issues I raised and provide insights and ideas towards my thesis. My thanks also goes towards Lesley Shannon, Mark Stoodley, Guy Lemieux, Paul Kundarewich, Maghsoud Abbaspour for their constant technical help and good friendship throughout these past two years. In addition, I would also like to thank Sean Peng for his encouragement and many words of advice.

I would like to thank two of my best friends from my undergraduate studies at U of T, Alice Lee and Karen Wang, for the encouragement and friendship they have shown me. A special thanks also goes to Alice for editing my thesis.

I would also like to thank my family and especially my sister Kate, for their ever-lasting love. As a really sweet sister, Kate constantly encourages me by bringing me lunches and helping me through late nights when I was working at school. I would also like to thank William Chow who has also been a great help to me with his input in many difficult technical problems and for faithfully walking me home every night.

Lastly, I would like to dedicate this thesis to these individuals and the many other friends I made during the six years of studies at U of T.

Contents

Abstract **i**

Acknowledgments **ii**

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**

1.1 Motivation 1

1.2 Objective and Contributions 2

1.2.1 Instruction Packing Algorithm 2

1.2.2 Stack Layout Optimization 3

1.2.3 Register Allocation 3

1.3 Thesis Organization 4

2 Background **5**

2.1 UTDSP Overview 5

2.1.1 Long Instruction Architecture 6

2.1.2 The Two-Level Memory System of UTDSP 7

2.1.3 The UTDSP Compaction Scheme 11

2.1.4 Summary 12

2.2 Compiler Techniques for Code Compaction 14

2.2.1 Procedural Abstraction and Live-range Recoloring . . . 15

2.2.2 Improved Procedural Abstraction and Live-range Recoloring 16

2.2.3 Summary 16

2.3 Hardware Approach to Code Compaction 17

2.3.1 The SPAM Project 18
2.3.2 BOPS Manifold Array 19
2.3.3 Summary 20
2.4 Graph Coloring and Register Allocation 21
2.4.1 Chow and Hennessy's Register Allocator 21
2.4.2 Genetic Algorithm for Graph-coloring 22
2.4.3 Summary 23
2.5 Trimaran's Infrastructure 23
2.5.1 Trimaran Compiler Overview 24
2.5.2 NYU Register Allocator 26
2.6 Summary 26

3 Benchmark Analysis 27
3.1 Operation Format and Categories 28
3.2 Operation and Long Instruction Analysis 30
3.3 Branch Offset Translation 33
3.4 Constant Analysis 34
3.5 Discussion 35
3.6 Summary 38

4 Methods for Achieving a Denser Decoder Memory 40
4.1 Achieving Compaction via Different Packing Techniques . . . 43
4.1.1 UTDSP Subset Packing Algorithm 43
4.1.2 UTDSP Subset Packing Algorithm with Binding-shifting and Commutative Transform 45
4.1.3 Packing Through Clique Partitioning 48
4.1.4 Divide and Conquer - Packing By Procedures 50
4.1.5 Results and Analysis 50
4.1.6 Summary 55
4.2 Compaction Achieved via Stack Optimization 56
4.2.1 Related Work in Stack Optimization 57
4.2.2 Trimaran's Original Stack Structure 57
4.2.3 Restructuring of Register Save Area 61
4.2.4 Register Spill Area Optimization 63
4.2.5 Implementing Stack Optimization for Other Compilers 64
4.2.6 Results and Analysis 67
4.2.7 Summary 69
4.3 Compaction Achieved via Coloring 70

4.3.1 Coloring with a Database 71
Original Color Selection Process 72
Pattern-aware Color Selection Process 73
Results and Discussion 77
4.3.2 Live-range Coloring Order 78
Results and Discussion 79
Summary 82
4.3.3 Live-range Coloring with Simulated Annealing 82
A Brief Overview of Approaches Attempted in Incorporating Simulated Annealing into the Allocator 83
The Graphical Formulation of a Live-range and the Operations Spanned by the Live-range 88
Recoloring the Live-ranges 93
Results and Discussion 94
Summary 101
4.4 Conclusion 102

5 Conclusions and Future Work **103**
5.1 Conclusions 103
5.2 Future Work 105

A **107**
A.1 Mapping to Clique Partitioning 107
A.2 Sizes of Each Individual Decoder Memory Bank 108

Bibliography **114**

List of Figures

2.1 A microprogram is encoded using nanoprogramming. 8
2.2 An example of a VLIW instruction where each field in the instruction contains an operation for the corresponding functional unit. 10
2.3 Subset packing and field Clustering. 13
2.4 UTDSP Compiler flow. 14
2.5 Two semantically identically code segments. 17
2.6 Trimaran compiler infrastructure. 25

3.1 Two Small VLIW Instructions as a Result of Clustering 31

4.1 (a) Instructions in their original format. (b) Storing the instructions in the two-level memory system. (c) Using bit mask to retrieve the original instructions. 42
4.2 The field with the least number of operations is assigned the highest rank. 44
4.3 (a) Using binding-shifting. (b) Using commutative transform. 48
4.4 Clique partitioning algorithm. 49
4.5 (a) Instructions are packed in sequential order. (b) Instructions with the highest costs are packed first. 54
4.6 (a) Trimaran stack organization. (b) The original register save area. (c) The modified register save area. 58
4.7 (a) Register Saving Code. (b) Register Restoring Code. . . . 62
4.8 (a) SUIF Stack Organization. (b) Register Saving Code. (c) Register Restoring Code. 65
4.9 Normalized Stack Sizes. 68
4.10 (a) A live-range spanning multiple operations. (b) r3 is selected, which allows operations, marked by asterisks, to be made identical to existing operations in the database. 72

4.11 Frequency-based color selection algorithm. 75
4.12 Opcodes listed in decreasing order of frequency. 76
4.13 Normalized Decoder Memory Sizes for the Database Optimization technique. 77
4.14 Scenarios that defeat the Color Selection Heuristic. 79
4.15 Normalized decoder memory size for different live-range coloring orders. 80
4.16 Spill code increase as a result of changing the order in which live-ranges are colored. 81
4.17 Spilling of two live-ranges to create identical operations. . . . 86
4.18 89
4.19 91
4.20 92
4.21 Algorithm: greedy recoloring technique. 95
4.22 Functions called by the main loop of the greedy recoloring algorithm. 96
4.23 Algorithm: recoloring with simulated annealing. 97
4.24 Functions called by the main annealing routine. 98
4.25 Unique operation pattern reduction. 99
4.26 100

List of Tables

3.1 The total unique operation patterns of the first cluster, categorized into three types. 32
3.2 The total unique operation patterns of the second cluster, categorized into three types. 32
3.3 All the unique VLIW instructions for the first cluster, categorized into multi-op or uni-op instructions. 33
3.4 All the unique VLIW instructions for the second cluster, categorized into multi-op or uni-op instructions. 33
3.5 Total unique operands and the total unique literal operands categorized in three types for cluster 1, before offset translation. 35
3.6 Total unique operands and the total unique literal operands categorized in three types for cluster 1, after offset translation. (S - stack related; B - branch related; O - other constants) . . 36
3.7 Total unique operands and the total unique literal operands categorized in three types, for cluster 2. 37

4.1 Subset Packing 50
4.2 Augmented subset packing using binding-shifting. 51
4.3 Augmented commutative subset packing using binding-shifting and commutative transform. 51
4.4 By Region 52
4.5 Clique Partitioning 52

A.1 Sizes of Each Decoder Memory Bank Required using Subset Packing 109
A.2 Sizes of Each Decoder Memory Bank Required using Augmented Subset Packing 110
A.3 Sizes of Each Decoder Memory Bank Required using Augmented Commutative Subset Packing 111

A.4 Sizes of Each Decoder Memory Bank Required using Packing by Region 112
A.5 Sizes of Each Decoder Memory Bank Required using Clique Partitioning 113

Chapter 1

Introduction

1.1 Motivation

Very long instruction word (VLIW) processors possess the ability to meet today's high computing demands from mobile and embedded applications to the next generation of enterprise computing platforms. In addition, the VLIW architecture is modular and scalable, which makes it well-suited for implementing application-specific programmable processors.

A VLIW processor consists of multiple functional units that are capable of executing independent operations in parallel with each other. Additional functional units with application-specific instructions can be easily added to the processor when required for performance. However, as multiple operations are directed to the functional units in one cycle, a long instruction is required to encode the operations. When a functional unit is idle in one particular cycle, a "non-operation" is encoded into the long instruction instead. Long instructions lead to a large instruction storage requirement, and the NOOPs create wasted space in the storage. Furthermore, high bandwidth is required to fetch the long instructions from an off-chip storage to the functional units on-chip, if off-chip storage is required. The high storage and bandwidth requirements have prevented VLIW architectures from being used in cost-sensitive systems.

1.2 Objective and Contributions

This thesis builds upon the two-level instruction packing and fetching mechanism developed by previous University of Toronto researchers. The objective is to explore different ways to achieve better code compaction under this framework, and ideally, without affecting performance.

The assembly code is generated by the Trimaran Compiler, a research compiler that is capable of generating long instructions for explicitly parallel-instruction architectures where the VLIW is a subset of this architecture. The characteristics of a set of SpecInt95 benchmarks, whose assembly code is generated by the Trimaran compiler, are analyzed and techniques to improve the code compaction are devised accordingly. The techniques examine three orthogonal directions to achieve code compaction and thus contribute to the major area of research for this thesis.

1.2.1 Instruction Packing Algorithm

Two techniques are proposed to enhance the existing packing algorithm. Both techniques focus on ways of transforming an instruction into equivalent instructions. An equivalent instruction can be used to replace the original instruction when the original instruction cannot be packed into an existing memory word but the equivalent instruction can. Hence, through increasing the opportunity for an instruction to be packed into an existing memory word, the storage requirement can be minimized and the memory utilization increased. The two techniques are able to achieve a maximum of 3.2% memory size reduction and a 100% utilization.

1.2.2 Stack Layout Optimization

Three techniques are proposed to restructure the stack to minimize the unique constant operands in memory operations that access the stack. Minimizing the unique constant operands in turn minimizes the number of unique operation patterns. One technique reorders the stack areas whereas the other two techniques focus on minimizing the size of the stack. As the unique stack related constants decrease, a maximum of 16.8 % in memory reduction is achieved. The latter two techniques are also applicable to other compiler systems and are especially useful for optimizing the stack layouts for embedded applications.

1.2.3 Register Allocation

Register allocation is manipulated to perform allocation in a way so as to reduce the unique operation patterns. Techniques to color and recolor live-ranges are proposed and tested. Furthermore, a novel idea of incorporating simulated annealing into graph coloring is attempted. Unfortunately, there are no significant reductions achieved by the techniques.

1.3 Thesis Organization

This thesis is divided into five chapters. Chapter 2 provides the reader with background information on the two-level memory system, various code compaction techniques, register allocation and a brief description of the Trimaran compiler. Chapter 3 provides a detailed analysis of the benchmarks used in the thesis. Chapter 4 describes the various compaction techniques proposed by the author. Finally, Chapter 5 concludes this thesis and offers recommendations for future work.

Chapter 2

Background

This chapter is divided into six sections. Section 2.1 provides an overview of the UTDSP project, from the microinstruction origin to the two-level memory system, and the various compression techniques developed by previous researchers. Section 2.2 examines other compiler assisted compression techniques developed at other universities. Section 2.3 describes a recent hardware approach to code compaction. Section 2.4 describes techniques for register allocation. Section 2.5 introduces the Trimaran compiler, the baseline compiler of this work. The last section, Section 2.6 summarizes this chapter.

2.1 UTDSP Overview

This section gives an overview of the UTDSP project.

2.1.1 Long Instruction Architecture

The ideas behind UTDSP's two-level memory system and its compaction scheme, discussed in section 2.1.2 and 2.1.3 respectively, originated from microprogramming in the 1970's [1]. This section provides a brief review of the different encoding schemes used in microprogramming.

In a microprogrammed computer, each machine instruction is executed by a microroutine consisting of one or more microinstructions. The microinstructions are stored in a special memory called the control store. As memory was an expensive resource in the 70's, different microinstruction encoding schemes were developed to minimize its storage requirement, which includes horizontal and vertical encodings, minimal encoding and nanoprogramming encoding [2].

In horizontal encoding, each bit of the microinstruction represents one control signal. Only a few control signals are active at one time, thus poorly using the available bit space. However, since multiple control signals can be active at the same time, parallel use of control functions are allowed.

In vertical encoding, each combination of useful control signals is enumerated, leading to a compact encoding. The disadvantage of this method is that more microinstructions are needed to perform the desired control functions and parallelism is not allowed.

The mostly widely used encoding, which is a combination of horizontal and vertical encodings, is called minimal encoding. Each control word is divided into a fixed number of fields that are each used to vertically encode a group of mutually exclusive micro-operations. The UTDSP's compaction scheme resembles this approach in that a bit mask horizontally encodes a

VLIW instruction and the operations belonging to one functional unit are vertically encoded into one operation field of the VLIW. A bit mask is a binary pattern of ones and zeros where a one indicates an operation for a function unit is being selected.

The last type of encoding to be discussed is nanoprogramming, which makes use of a nanostore to hold nanoinstructions. The concept of nanoprogramming is illustrated in Figure 2.1. If the size of each microinstruction is w bits and the microprogram contains n microinstructions, then a control memory of size $w \times n$ bits would be required. However, if only m of the microinstructions in the microprogram are unique, a smaller control store (called a nanostore) of size $m \times w$ could be used to hold the m unique microinstructions. Each microinstruction in the original program could then be replaced by the address of the word in the nanostore that contains the appropriate microinstruction. The number of bits in an address for the example illustrated in Figure 2.1 can be easily computed as $k = \log_2 m$ where m is the number of unique microinstructions. This approach is what gives rise to UTDSP's two-level memory system.

2.1.2 The Two-Level Memory System of UTDSP

The UTDSP architecture is a very long instruction word (VLIW) architecture consisting of multiple functional units. Each unit is capable of executing an operation independently from other functional units.

The operations feeding the functional units at one cycle are scheduled in parallel to form a long instruction. An instruction, as illustrated in Figure 2.2, contains seven fields and each field controls one of the seven functional

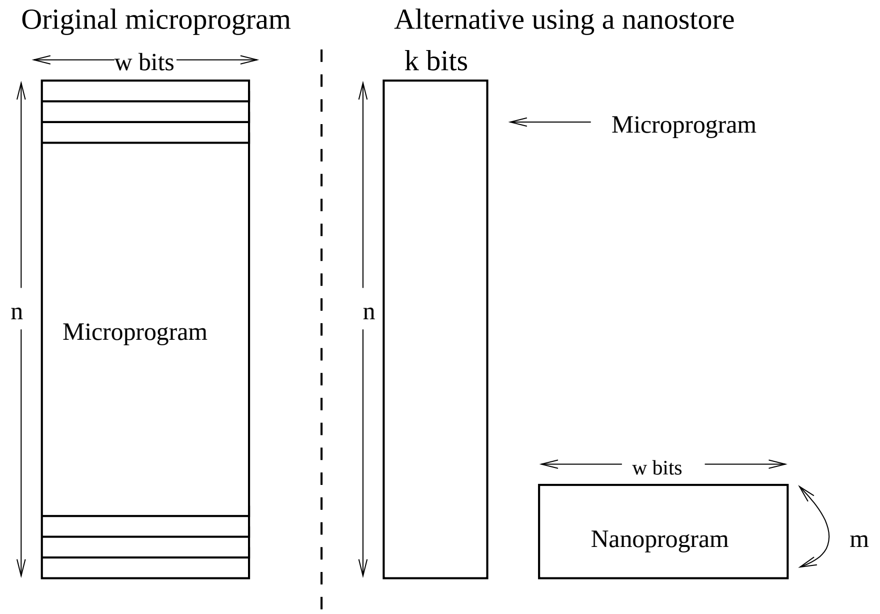

Figure 2.1: A microprogram is encoded using nanoprogramming.

units of a VLIW. The seven units in the VLIW consist of two integer units, two floating-point units, two memory units and one branch unit. Since the number of operations that can be scheduled in one cycle is often less than the number of functional units, NOOPs are inserted into the fields that do not have operations scheduled in that particular cycle. As a result, the storage space for a VLIW processor is poorly used since much space is wasted in storing the NOOPs. To minimize the storage space, a two-level instruction memory system, similar to that of a nanostore, was introduced by Mazen Saghir [1].

The two-level memory system consists of two blocks: the instruction memory and the decoder memory. Figure 2.2 shows the two-level memory system used in UTDSP. Instructions that contain only one operation, the uni-op instructions, are stored in the instruction memory, while the other instructions, the multi-op instructions, are stored in the decoder memory in a packed format. The multi-op pointers, which are the addresses of the multi-op instructions in the decoder memory, are stored in the instruction memory along with the uni-op instructions. When a word is fetched from the instruction memory, its most-significant bit is examined to determine if the word is a uni-op operation or a multi-op pointer. If it is a uni-op operation, it is directly issued to an appropriate functional unit where it can be executed. On the other hand, if a multi-op pointer is fetched, it is used to access the memory locations in the decoder memory banks. Then the set of operations stored in the locations addressed by the multi-op pointer will be dispatched to their corresponding functional units. There are seven functional units in UTDSP as shown in Figure 2.2.

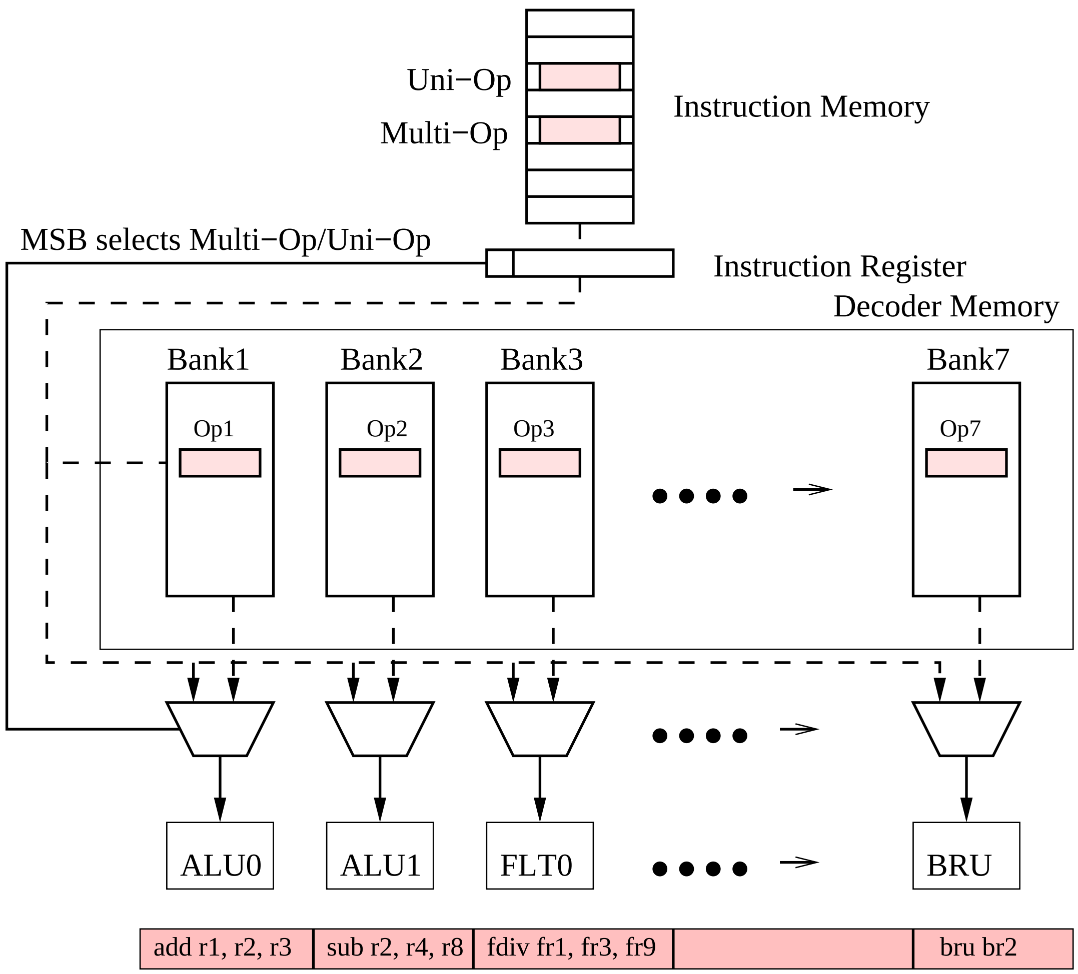

Figure 2.2: An example of a VLIW instruction where each field in the instruction contains an operation for the corresponding functional unit.

2.1.3 The UTDSP Compaction Scheme

It was then realized that when the long instructions are stored in the decoder memory in the original format, empty slots are created as a result of the NOOPs in the instructions. As illustrated in Figure 2.3 (a), twelve empty slots are created when three long instructions are stored in the original format in the decoder memory.

Hence, M. Saghir then proposed a scheme to pack the long instructions in a way similar to minimal encoding. This method is bit masking and is illustrated in Figure 2.3 (b). The instructions originally stored in locations 0x30 and 0x20 can be pack into the same location and bit masks are used to restore them back to the original format. The bit masks are also stored in the instruction memory along with the addresses.

M. Saghir and S. Peng further fine-tuned the packing scheme by introducing the ideas of field clustering and subset packing. As shown in Figure 2.3 (c), the instructions are divided into two halves, or two clusters. The first cluster contains fields B1 to B3, where as the second cluster contains fields B4 to B7. The instructions in the first cluster can be packed into two locations and the instructions in the second cluster can be packed into one location. The saving in decoder memory for the second cluster is a result of subset packing where identical operations can be represented by one unique copy. Both ideas are further illustrated in Chapter 4.

Since subset packing largely depends on the order in which instructions are considered, they also proposed the usage of an $O(n^3)$ algorithm. The main functions of the algorithm are: to rank the fields, to assign a priority to each instruction, to divide the instructions into complementary sets, to pack instructions according to the priorities, and finally to update the ranking and priorities of the fields and the instructions. Details of the algorithm are summarized in Section 4.1.1.

The algorithm is able to achieve denser packing as it attempts to push the instructions consisting of more fields or of higher ranking fields into the bottom, or the lower addresses, of the decoder memory bank. The main disadvantage to this algorithm is that it is slow when a large benchmark is considered.

The compiler flow for the UTDSP project is shown in Figure 2.4. A C program is parsed and analyzed by the front end of the Trimaran compiler, also referred to as Impact, where Instruction-level parallelism (ILP) optimizations are performed. Then, the operations are passed to the back end of the compiler, also referred to as Elcor, where instruction scheduling and register allocation are performed. The long instructions are then passed to the UTDSP packer, where compaction takes place. The infrastructure of the Trimaran compiler is further explained in Section 2.5.

2.1.4 Summary

This section provides an overview to the UTDSP project, focusing on the two-level memory system and the compaction scheme that forms the basis for the compaction work carried out in this thesis.

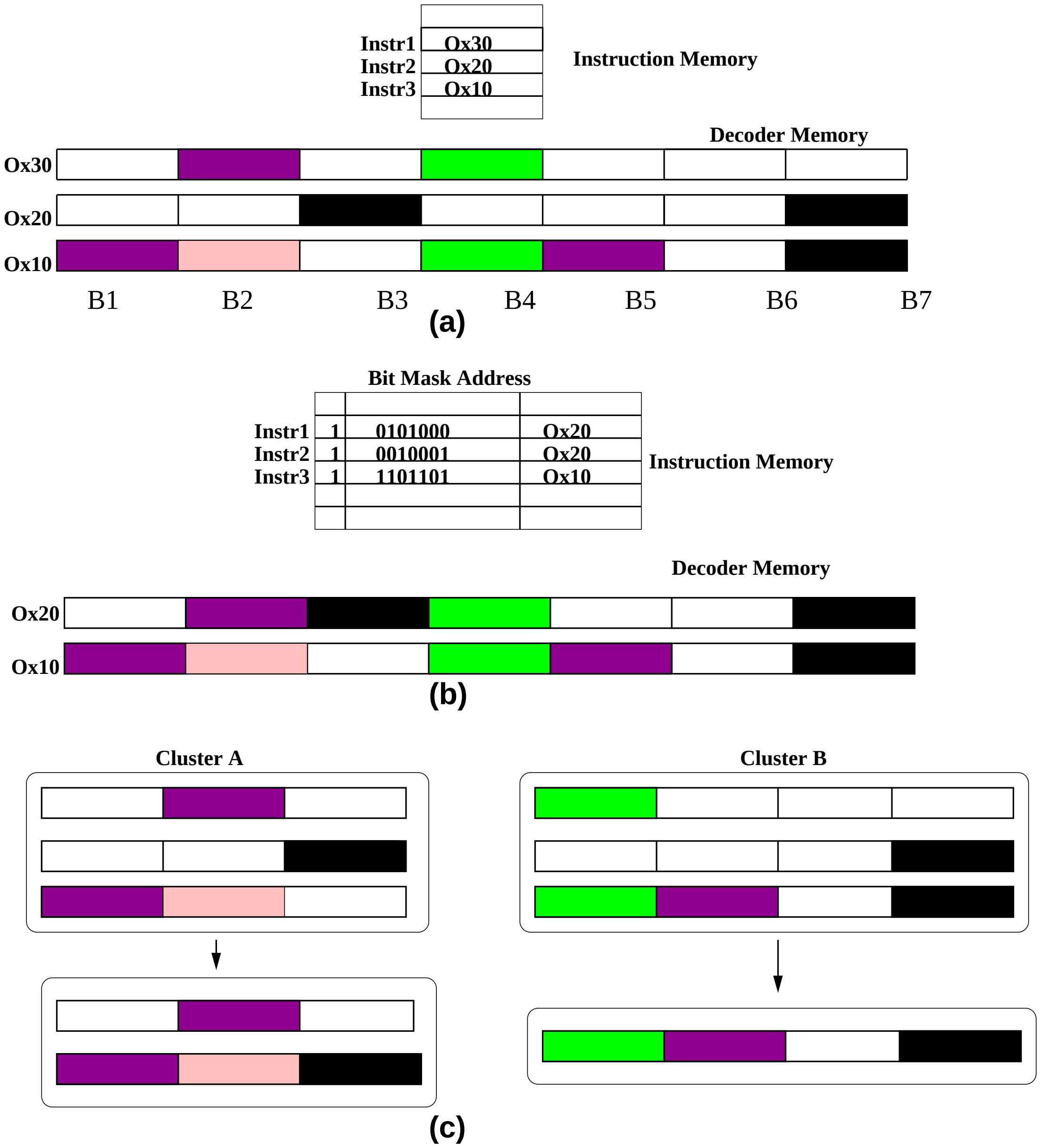

Figure 2.3: Subset packing and field Clustering.

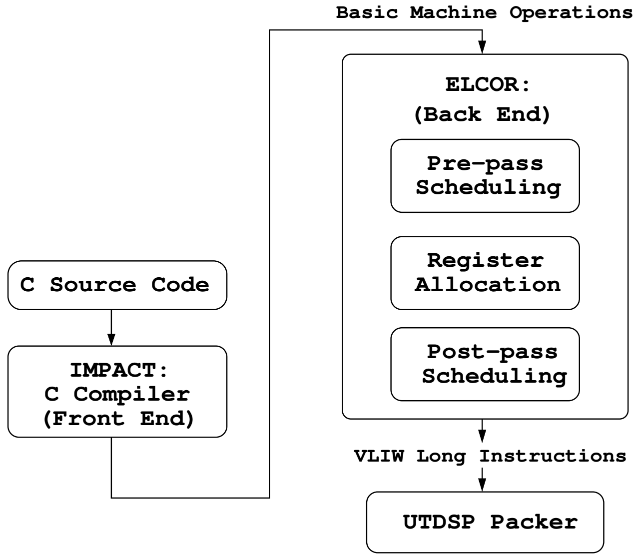

Figure 2.4: UTDSP Compiler flow.

2.2 Compiler Techniques for Code Compaction

As one aspect of the thesis examines the use of register allocation to enhance code compaction, it is worthwhile to examine a few compiler-assisted code compaction techniques developed by other researchers.

The two most notable works were done by S. Debray et al. [3] and K. Cooper [4]. With their proposed techniques, K. Cooper was able to achieve an improvement of 1% to just under 5% for a set of C programs designed for signal processing and for voice/data/video coding and compress. S. Debray et al. was able to to achieve 30% code compaction. Though both Debray and Cooper's approaches are based on an orthogonal framework, different from that of UTDSP, the approaches are still novel and worth mentioning.

2.2.1 Procedural Abstraction and Live-range Recoloring

K. Cooper's work proposes two approaches. The first approach is to identify the "repeats" of the assembly code of a program and perform procedural abstraction or cross jumping on the repeated segments.

To identify the "repeats", a suffix tree [5] is built where each unique line of code is an alphabet. After each line is translated into an alphabet, a string of text is formed. This string of text is passed into a suffix tree engine in which all the repeated segments can be identified [6]. In procedural abstraction, one repeated segment is made into a procedure and any subsequent repeats are replaced by a call to that procedure. In cross jumping, a new label is inserted into one repeated segment and any subsequent repeats, are replaced by a jump to that label. However, all the out-branches in each segment must match in order for cross-jumping to be applied. Both techniques give a joint effort of less than 1% code size reduction.

The second approach is live-range recoloring, also called register renaming. This approach is preceded by a register abstraction analysis in which two nearly identical code segments are identified, i.e., differing only in register assignments. An attempt to make these near-identical code segments identical is done by recoloring the live-ranges. This approach is able to give a reduction just under 5%.

There are a few problems associated with both methods. Both approaches are accompanied by a roughly equivalent increase in dynamic instruction count, which affects performance. Also, although recoloring allows the abstraction of one particular pair of code segments, the recolored live-range may interfere with the abstraction of a different pair of code segments.

2.2.2 Improved Procedural Abstraction and Live-range Recoloring

S. Debray's approach is also very similar to Cooper's previously described approaches. He also abstracts similarly coded segments through register renaming. However, instead of renaming the entire live-range, he renames the register within one basic block and inserts register moves at the entry and exit points of the basic block. Debray proposes that the procedural abstraction of code should be done under the scope of a basic block. He is able to achieve 10% code size reduction using this approach.

The architecture specific reductions come from the abstraction of the register saving code at function prologs and epilogs and the elimination of redundant computations of the global pointer register, gp. This results in a 15% code size reduction.

The other 5% reduction comes from other inter-procedural optimizations and useless code elimination through his binary code optimizer.

2.2.3 Summary

Both Debray and Cooper are able to achieve code size reduction via procedural abstraction. Prior to performing procedural abstraction, repetitive code segments are identified and near-identical code segments must be made identical through live-range recoloring. Cooper identifies identical code segments using a suffix tree approach while Debray uses a finger printing approach.

```
r1 = r2 + r3
r4 = r2 - r3
r5 = r1 + r4
```
(a)

```
r4 = r2 - r3
r1 = r2 + r3
r5 = r1 + r4
```
(b)

Figure 2.5: Two semantically identically code segments.

However, both identification approaches have an inherent disadvantage, that is, they are unable to identify semantically identical yet lexically un-identical code segments. Both identification approaches can be easily defeated by two semantically identical code segments illustrated in Figure 2.5 (a) and (b).

The UTDSP approach is novel in that it overcomes the instruction ordering problem. The subset packing of instructions in the decoder memory allows instructions with identical operations to pack with each other, regardless of the ordering in which the instructions appear in the original program.

2.3 Hardware Approach to Code Compaction

After examining software approaches to code compaction, it is worthwhile to examine some hardware approaches to achieve code reduction for VLIW processors. Approaches taken by the SPAM project and BOPS Manifold Array are summarized in this section.

2.3.1 The SPAM Project

The SPAM project proposes two methods of achieving code compaction based on the *external pointer macro* (EPM) model [7]. In this model, a compressed program consists of a dictionary and a skeleton. The dictionary contains code segments that occur frequently, or in other words, repeat themselves in the original program. The skeleton contains instructions from the original program, interspersed with pointers to the dictionary. The two methods differ in how the dictionary and pointers are represented.

The first method is a software approach where no additional hardware support is required. One copy of the repeated code segments is appended with a *RET* instruction and placed into the dictionary. Any occurrences of this code segment in the program is substituted with a *CALL* instruction. When the *CALL* instruction is executed, the address of the next instruction is placed in the link register and the control is transfered to the destination address in the dictionary where the repeated code segment is stored. When the *RET* instruction at the end of the code segment is reached, control is then transfered back to the address stored in the link register.

The second method is a hardware approach which requires additional hardware such as a counter. In this approach, one copy of the repeated code segments is placed into the dictionary but a *RET* instruction is not appended to the end of the segment. Any occurrences of this code segment in the program is substituted with a *call-to-dictionary* or a *CALD* instruction. The *CALD* instruction is similar to a *CALL* instruction as it also stores the return address into the link register and transfers control to the designated location in the dictionary. Furthermore, the *CALD* instruction sets the counter to a number which indicates the number of instructions in the dictionary that remain to be executed. The counter counts down towards zero as each instruction is executed. When the count reaches zero, control is automatically transfered back to the address stored in the link register. Hence, a *RET* instruction is not needed in the second method.

An algorithm which identifies meaningful repeated code segments is also proposed. The software and hardware methods are able to achieve a code size reduction of 12% and 16% respectively, when compared with the uncompacted code generated by TI's TMS320C25 compiler.

2.3.2 BOPS Manifold Array

The BOPS Manifold Array [8] is one that also uses a two-level memory system to achieve code compaction, but in a much different context than that of the UTDSP. It is a SIMD and MIMD combined architecture consisting of an array of processing elements (PE). Each PE consists of five functional units and a small memory called the indirect-VLIW Instruction Memory (VIM). The functional units are the arithmetic unit, the store unit, the load unit, the data select unit, and the multiply/divide unit.

There are two instruction formats: a non-compacted 32-bit instruction and a compacted instruction that contains two 15-bit compacted instructions, each of which can be translated back into a 32-bit instruction. Once an instruction word is dispatched to all the PEs through the 32-bit instruction bus, each PE needs to determine whether it is a compacted or non-compacted instruction. The PE then translates it, decodes it, and executes the instruction on its functions units.

There are two forms of the 15-bit compacted instruction and each is translated back to the 32-bit instruction via one of the following methods: the fixed direct translation and the dynamic VIM translation.

In a fixed direct translation, a local register on the PE is pre-loaded with bits that can be concatenated to the operand bits of the compacted instruction, allowing the instruction to use the full architecture register space. In dynamic VIM translation, a compacted instruction provides a VIM address that selects a 32-bit instruction template previously loaded by the programmer into the VIM. The compacted instruction supplies the frequently changed fields, such as the register operands. By plugging the compacted instruction into the template, a 32-bit instruction is formed.

Since a 32-bit instruction can be translated into two 32-bit instructions, two of the functional units in the PE can execute them in parallel, providing VLIW performance.

2.3.3 Summary

BOPS ManArray provides a way to translate an instruction into two instructions, using another level of memory. However, the VLIW instruction is limited to two operations and the utilization of the functional units in a PE is low.

The EPM model proposed in the SPAM project is also similar to the two-level memory system in UTDSP. However, the way in which the repeated instructions are stored in the dictionary and how the pointers are represented are significantly different from the bit masking idea used in UTDSP. The UTDSP bit masking approach provides a much more flexible framework for instruction packing. Also, the proposed identification algorithm cannot identify semantically identical, yet lexically different code segments.

2.4 Graph Coloring and Register Allocation

Register allocation has been extensively studied and improved to minimize the number of memory references. As a result, register allocation may lower the instruction count as well as reduce the execution time per instruction by changing memory operands to register operands. As one aspect of this thesis seeks ways to perform "smart" allocation, it is worthwhile to review some important ideas behind register allocation as well as describe a few orthogonal approaches to graph-coloring.

2.4.1 Chow and Hennessy's Register Allocator

Register allocation has been mapped into a graph-coloring problem, where each live-range is represented by a node. An edge connects the nodes if two live-ranges overlap, or interfere. The idea is to color the graph such that no two nodes connected by an edge is assigned the same color [9]. Minimizing the number of memory references entails minimizing the register saving code generated as a result of call overhead and the spill code generated as a result of an uncolorable graph. The order in which live-ranges are selected and colored determines the amount of spill code generated while the linkage convention determines the amount of register saving code generated. Chow et al. proposed a priority based coloring scheme such that each live-range is assigned a priority. The priority is an empirical measure of the relative benefits of assigning a given live range to a register. The priority is proportional to the total amount of execution-time savings gained due to the live-range's residence in a register. The live-ranges with higher priorities are colored first.

Chow el al. also proposed a linkage convention where the machine registers are divided into two fixed sets: caller-saved and callee-saved registers. This convention provides extra opportunities to minimize call overhead. For instance, a procedure that does not make any other call (a leaf procedure) should favor the use of caller-saved registers, since their previous contents do not have to be saved before they can be used. If the body of a procedure contains many calls, callee-saved registers are more preferable because saving and restoring once at the procedural entry and exit, respectively, is cheaper than saving before each procedural call and restoration after the call.

2.4.2 Genetic Algorithm for Graph-coloring

There are also attempts to solve the graph-coloring problem using heuristic approaches such as genetic algorithms (GA), where a constraint, such as the least number of colors, is imposed. Sami et al. proposed using grouping GA [10] to solve the coloring problem while Grewal [11] proposed another version of GA, called the *enhanced GA* to solve the problem. However, neither studies provided execution speed for large testcases with more than 1100 edges, or more than 50 nodes. Furthermore, both studies isolate the graph-coloring problem from the register allocation context where issues such as the number of memory references are not addressed. Also, realistic applications are likely to have more than 50 live-ranges and more than 1100 interference edges. The long time requirement for coloring a large graph makes GA a non-practical approach.

2.4.3 Summary

Register allocation using heuristics such as GA or simulated annealing (SA) is a creative idea. However, there has been much study and many well established methods on the performance of the coloring such that the efficiency of the colored code is optimized, i.e., the spilled code is minimized. If GA or SA are to be used to produce an efficient coloring, it must be carefully guided so as to obey the live-range selection rules and linkage convention established by previous researchers. Although GA or SA can easily accept other optimization constraints, such as minimizing the operand patterns, obeying the selection and linkage conventions already limit its search space by a great deal.

2.5 Trimaran's Infrastructure

The next major boost of microprocessor performance is in instruction-level parallel processing. While mainstream microprocessors in 1990 executed one instruction per clock cycle [12], hardware technology is expected to produce microprocessors that can execute up to 16 instructions per clock cycle. Such improvements in hardware parallelism have placed a tremendous pressure on the compiler technology. The quality of compiler parallelization techniques can potentially make an order of magnitude difference in program execution performance for processors that exploit ILP. For a processor that executes up to eight instructions per cycle, a well-parallelized program can run at several times the speed of a poorly parallelized version of the same program.

As the UTDSP project progressed to a certain stage, it also needed to face the possibilities of incorporating more advanced compiler techniques such as predication and speculation into its compiler, to exploit its significant hardware parallelism. It is with this view that the Trimaran compiler is introduced into the UTDSP project.

2.5.1 Trimaran Compiler Overview

Trimaran is a compiler conceived for research in instruction-level parallelism (ILP). Its main purpose is to serve as a vehicle to investigate processor architectures having significant parallelism and to investigate the compiler technology needed to effectively exploit such architectures [13]. It is written as a joint effort by the researchers at the University of Illinois Urbana and Hewlett Packard. The compiler consists of three main components, the front-end, the back-end and a simulator. Figure 2.6 illustrates the components of the Trimaran compiler.

The front-end of Trimaran is also known as Impact, which parses C code, admits an architectural independent intermediate representation, and performs various classical optimizations. Furthermore, it performs ILP optimization. For example, it merges basic blocks along a frequently executed path, forming a trace or a superblock. Also, it performs predicated transformation (also called if-conversion) forming hyperblocks by removing the performance limitations imposed by the control decision of the if-else-then clause [14]. Both hyperblocks and superblocks are regions that are large enough to potentially contain many independent instructions for code

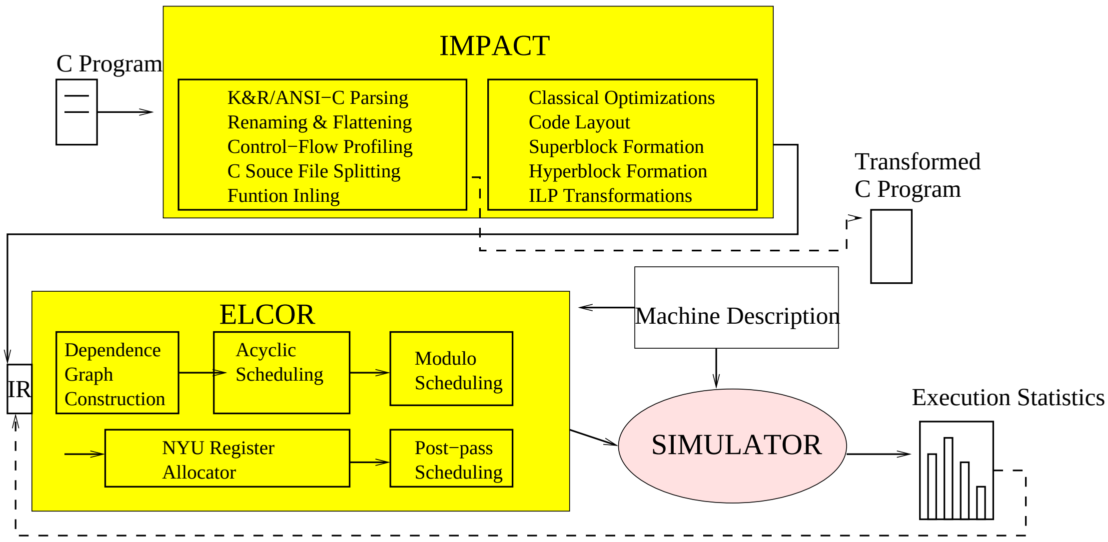

Figure 2.6: Trimaran compiler infrastructure.

scheduling so that processors may execute them in parallel.

In addition, Impact also performs speculative transformation. Control speculation refers to breaking inherent programmatic dependences of branches by guessing the outcome of a run-time event at compile time. A control dependence is broken by guessing that a branch will go in a particular direction, thereby making an operation's execution independent of the branch.

Data speculation allows the compiler to aggressively reorder memory operations and effectively reduce memory dependence height, which results in a more compact schedule.

The back-end of Trimaran performs mainly the task of code scheduling and register allocation.

The Trimaran opcode repertoire, at its core, is similar to that of UTDSP, which is a RISC-like load/store architecture, with standard integers, floating-point, and memory operations. In addition, it has operations to support speculation, e.g., LDS for a speculative load operation and CMPP for a compare-to-predicate operation.

2.5.2 NYU Register Allocator

The Trimaran register allocation pass is written by a Ph.D. student at New York University [15]. It is an allocator based on Chow's allocator with enhanced priority functions and techniques addressing hyperblocks and superblocks. As both blocks are much larger than basic blocks, certain aspects such as live-range granularity need to be reconsidered. Also, the NYU allocator attempts to make use of profile information to decide on the splitting point of a live-range, when there is a shortage of registers.

However, the Trimaran release was not complete and only Chow and Hennessy allocator without life-range splitting aspect is working. This is the baseline allocator used in this thesis.

2.6 Summary

This chapter has provided an overview to the UTDSP project, examined recent approaches to code compaction, reviewed key concepts of register allocation and introduced the Trimaran compiler.

Chapter 3

Benchmark Analysis

This chapter provides a detailed analysis of the assembly code generated by the baseline compiler for a set of specInt95 [16] benchmarks. This set of benchmarks, which includes 129.compress, 099.go, 130.li, 124.m88ksim, 134.perl and 147.vortex, is used to evaluate the effectiveness of the compaction techniques developed in Chapter 4. The six benchmarks used are compiled with the following compilation options respectively: hyperblock, basic block, hyperblock, hyperblock, hyperblock, and basic block. Ideally, all benchmarks should be compiled using the hyperblock option for obtaining the highest ILP. However, basic block options are used for benchmarks go and vortex because they could not be successfully compiled using the hyperblock option. The other two SpecInt benchmarks are omitted due to memory and compilation time constraints. Various aspects of the assembly code are also examined such as the operand, operation and instruction patterns. By gathering statistics on numbers of unique operand, operation and instruction patterns, characteristics of the benchmarks can be observed, giving insights to ways of minimizing the patterns.

This chapter is divided into five sections where the first four sections focus on methods for generating the data. Observations and analyses are collectively presented in the last two sections. The first section, Section 3.1, introduces the operation format and a scheme in which an individual operation is categorized. Section 3.2 provides statistics on the operation and instruction patterns of each benchmark. Section 3.3 discusses one particular operation, the prepare-to-branch-relative operation. Section 3.4 examines the operands of each operation, focusing on the literal operands. Section 3.5 discusses the results presented in Sections 3.2 to 3.4. Finally, Section 3.6 summarizes this chapter.

3.1 Operation Format and Categories

An *operation* corresponds to an elementary operation performed by a functional unit and a *VLIW instruction* or an *instruction* consists of several operations. Though an operation or an instruction is meant to be executed on the real hardware, Trimaran uses a simulator, instead of a real processor, to obtain run-time results. Hence, the instruction set of Trimaran, as it is only executed by a simulator, does not provide a real definition at the "bit encoding" level. An example of such an operation is the relative branch operation. Offsets for relative branch operations do not correspond to the actual physical address offsets. Instead, labels of the target addresses, which can be understood by the simulator, are used as the offsets.

The VLIW architecture assumed by the simulator and used as the baseline archichitecture of the thesis, consists of two integer units, two floating point units, two memory units and one branch unit. This means, an instruction consists of seven operations as was proposed for UTDSP.

An operation specification consists of the following [13]:

1. An opcode.

2. A list of source operands.

   A source operand is either a literal or a register specifier. A literal specifier can be a label or an immediate operand such as a constant. A register specifier can be the name of a macro register, or a colorable register.

   Each register file has a unique name. A register specifier consisting of the name of the register file and a register number within the register file is referred to as a colorable register. The Trimaran compiler reserves a few registers for special purposes such as parameter-passing, spill address calculation and stack manipulation. They are referred to as macro registers.

3. A list of destination operands.

   A destination operand is a register specifier. Certain operations have no destination operands, such as the case of store operations and some branch operations. Certain operations have more than one destination operand as in the case of predicated operations.

Each operation can be categorized into one of following three types:

1. Type I.

   An operation where all its operands are colorable registers.

   e.g. GPR1 = ADD.W (GPR2, GPR3) ;GPR1 $\leftarrow$ GPR2 + GPR3

2. Type II.

   An operation where at least one, but not all, of its operands are colorable registers.

   e.g. GPR1 = ADD.W (GPR2, #3) ;GPR1 $\leftarrow$ GPR2 + 3

3. Type III.

   An operation where none of its operands are colorable registers. The operands are either macro registers or constant literals.

   e.g. TEMP_REG = ADD.W(SP, #218) ;TEMP_REG $\leftarrow$ SP + 218

   TEMP_REG and SP are macro registers reserved for the purpose of live-range spilling and stack manipulation, e.g., SP is the stack pointer.

3.2 Operation and Long Instruction Analysis

M. Saghir and S. Peng demonstrated the effectiveness of using the field clustering technique to achieve a denser decoder memory packing (Section 2.1.3). They chose the number of clusters to be two, which was determined to provide the best trade-off between denser packing and the degradation of performance resulting from a smaller size of decoder memory. As this thesis

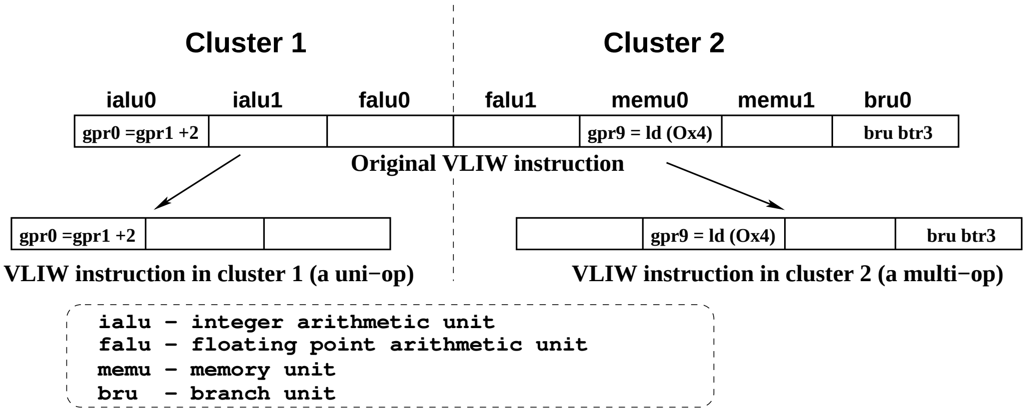

Figure 3.1: Two Small VLIW Instructions as a Result of Clustering

adopts the idea of field clustering with two clusters, each VLIW instruction can be viewed as two small VLIW instructions, one that consists of three operation slots and the other consisting of four operation slots. Each VLIW can be further categorized into a multi-op or an uni-op instruction, depending on the number of operations contained in the instruction. For example, the VLIW instruction in cluster 1 of Figure 3.1 is uni-op instruction whereas the VLIW instruction in cluster 2 is a multi-op instruction.

Since the instructions in each cluster will be packed independently from one another, two different sets of data are shown. Tables 3.1 and 3.2 show the total operation counts and the total unique operation patterns categorized into three types for each cluster, respectively. For example, Table 3.1 indicates that there are in total 5020 operations in benchmark compress and 3186 of which are unique. Hence, 1834 of the total operations are repeated patterns of the unique operations in cluster 1. Tables 3.3 and 3.4 show the total VLIW counts and the total unique VLIW patterns categorized into multi-op and uni-op instructions for each cluster. These data are generated using the assembly code compiled by Trimaran and counting each type of operations or instructions accordingly.

Benchmarks	compress	go	li	m88ksim	vortex
Total Operations	5020	55548	18495	40952	139625
Unique Type I	527	2316	314	1603	1467
Unique Type II	2119	18439	5306	14796	29915
Unique Type III	540	1211	798	1860	3573
Total Unique Operation Patterns	3186	21966	6418	18259	34955

Table 3.1: The total unique operation patterns of the first cluster, categorized into three types.

Benchmarks	compress	go	li	m88ksim	vortex
Total Operations	3332	34534	13427	25559	99128
Unique Type I	53	25	133	241	35
Unique Type II	920	1974	1201	2934	3850
Unique Type III	5	9	7	19	31
Total Unique Operation Patterns	978	2008	1341	3194	3916

Table 3.2: The total unique operation patterns of the second cluster, categorized into three types.

The analysis and the insights drawn from these data are presented in Section 3.5.

Benchmarks	compress	go	li	m88ksim	vortex
Total VLIWs	3171	38332	12423	25533	90720
Unique Multi-op	1665	15208	5110	13369	30614
Unique Uni-op	859	6625	1599	3747	8040
Total Unique VLIW Patterns	2524	21833	6709	17116	38654

Table 3.3: All the unique VLIW instructions for the first cluster, categorized into multi-op or uni-op instructions.

Benchmarks	compress	go	li	m88ksim	vortex
Total VLIWs	2541	29997	11422	20541	77290
Unique Multi-op	454	1558	790	2136	3759
Unique Uni-op	440	1414	822	1664	2150
Total Unique VLIW Patterns	894	2972	1612	3800	5909

Table 3.4: All the unique VLIW instructions for the second cluster, categorized into multi-op or uni-op instructions.

3.3 Branch Offset Translation

The branch operation is split into two operations: a prepare-to-branch (PBR) operation and the actual branch operation. The PBR operation loads the target address, or the offset as in the case of PBR-relative (PBRR) into a register. Then the branch operation simply branches to the address specified by the register. This branch architecture allows the compiler to schedule the PBRR operation well in advance of the actual branch operation, allowing the information to be provided to the branch hardware in a timely manner.

An example of the PBRR operation generated by the compiler is as follows:

```
PBRR [br<btr 2>] [b<25> i<0>] p<t>
```

This operation loads the offset into BTR2, which is a branch target register. The offset is calculated as the difference between the address of the long instruction containing the PBRR operation and the address of the first instruction in block 25 of the current procedure. However, as shown in the example, the value that is loaded into BTR2 is a label, not the actual offset constant. The reason was briefly mentioned in Section 3.1. Since a simulator is used to execute the instructions generated by Trimaran, a label b<25> is sufficient for the simulator to interpret the operation and determine the target address. Therefore, to obtain a proper analysis of the distribution of literals, the labels need to be translated into the real offset constants. Assuming that each operation is four bytes long, the offsets can be easily computed.

3.4 Constant Analysis

There are many unique literal operands in the benchmarks. They can be categorized into three types: stack-related literals, branch-related literals, and others. The branch-related literals can be labels used for absolute branching, or offsets. The others can be constants from the source program, or constants generated during optimization passes of the compiler, e.g., rematerialization[1].

[1]Rematerialization is a spill reduction technique of replacing spills and reloads with instructions that recompute values that have been evicted from registers using constants which are never-killed [17].

Table 3.5 and Table 3.6 provide a constant analysis for cluster 1, before and after offset translation. An analysis of the data is provided in Section 3.5.

Benchmarks	compress	go	li	m88ksim	vortex
Total Unique Operands	908	2362	1326	3002	5336
Unique literal Operands					
Stack	448	860	431	998	1595
Branch	124	782	373	802	1205
Others	256	618	413	1089	2602
Total	723	2187	1183	2764	5202

Table 3.5: Total unique operands and the total unique literal operands categorized in three types for cluster 1, before offset translation.

Table 3.7 provides a constant analysis for cluster 2. The data are identical for before and after translation as there are no PBR operations in cluster 2.

Tables 3.6 and 3.7 also provide cumulative percentages of the total number of literals that can be represented by 4, 8 12, 16, 24 and 32 bits. For example, Table 3.6 indicates that 24% of the branch related literals for benchmark 129.compress can be represented in 8 bits, i.e. having values less than 128. Knowing the distribution of the constants helps in estimating the number of bits required for representing an operation in the bit encoding level.

3.5 Discussion

Table 3.1 and 3.2 show that type II operations are the most commonly generated type of operation by the baseline compiler. Although the Type II operation forms the majority of all operations, their patterns are difficult to minimize because not all the operands of a type II operation are colorable

Benchmarks	compress	go	li	m88ksim	vortex
Total Unique Operands	1213	9448	3530	7067	22756
Unique literal Operands					
Stack	488	860	431	998	1595
Branch	527	8134	2694	5320	18938
Others	256	618	413	1089	2602
Total	1028	9273	3387	6829	22622
Total Number of Bits	Cumulative % of Literals				
	S-B-O	S-B-O	S-B-O	S-B-O	S-B-O
4 bits	2-19-64	13-6-81	24-10-92	24-4-74	33-7-88
8 bits	15-**24**-92	22-6-88	45-11-99	35-4-92	45-7-96
12 bits	98-64-99	98-9-93	98-22-99	96-15-96	97-8-99
16 bits	100-100-99	100-93-100	100-91-100	100-91-98	100-31-99
24 bits	100-100-99	100-100-100	100-100-100	100-100-99	100-100-99
32 bits	100-100-100	100-100-100	100-100-100	100-100-100	100-100-100

Table 3.6: Total unique operands and the total unique literal operands categorized in three types for cluster 1, after offset translation. (S - stack related; B - branch related; O - other constants)

registers. On the other hand, all the operands of type I operations are colorable registers but, they only occupy a small percentage of the total unique operation patterns. Thus, minimizing the unique type I operation patterns will not lead to a significant overall pattern reduction.

The ratio of total unique operations to total operations in cluster 2 is significantly less than that of cluster 1. In other words, there are more repeated operation patterns in cluster 2 and thus those operations are more “packable”. The main reason is that there are significantly fewer literal

Benchmarks	compress	go	li	m88ksim	vortex
Total Unique Operands	172	212	127	244	756
Unique literal Operands					
Stack	0	0	0	0	0
Branch	0	0	0	0	0
Others	13	45	7	38	444
Total	13	45	7	38	444
Cumulative % of Literals	Others	Others	Others	Others	Others
4 bits	95	74	98	95	87
8 bits	95	78	99	97	87
12 bits	99	82	100	98	95
16 bits	99	100	100	99	99
24 bits	100	100	100	99	99
32 bits	100	100	100	100	100

Table 3.7: Total unique operands and the total unique literal operands categorized in three types, for cluster 2.

operands for cluster 2 operations, as can be seen from Table 3.7. In addition, since the ratio of total unique operations to total operations is much higher in cluster 1 than in cluster 2, cluster 1 also has a higher ratio of total unique VLIW instructions to total VLIWs, as can be seen in Tables 3.3 and 3.4.

Comparing Tables 3.5 and 3.6, the process of offset translation also increases the total number of literals dramatically, leading to the increase of total unique operands. As future work, different ways to perform code layout can be explored to reduce the total number of literals related to PBRR operations. Table 3.6 also shows that the values of offset literals resulting from the translation process are large and in the benchmark of m88ksim, 24 bits are required to represent some of them. The large offset constants are unconventional as compared with the offset values of a traditional load-store machine, which are generally much smaller [18]. In contrast to the relative branchings of a traditional load-store machine, which are often to targets that are 4 - 7 instructions away, the relative branch operations in Trimaran are also used to branch to another function, similar to an absolute branching, thus, resulting in large offset values.

Another observation that can be made about literal operands from Tables 3.5 and 3.6 is that they form the majority of the total unique operands. For example, in Table 3.5, for benchmark 129.compress, there are 908 unique operands, 723 of which are unique literals. These unique literal operands are the main cause for the many unique operation patterns for cluster 1, hence making the operations less packable.

3.6 Summary

This chapter examines the benchmarks in terms of operand, operation and instruction patterns. The key observations can be summarized in three points. First, unique type II operations form the majority of the total unique operation patterns. Second, unique constant operands form the majority of the total unique operand patterns. Third, the unique constant operands are the main cause for the large number of unique operations.

From the above observations, it is evident that unique operation patterns cannot be minimized without minimizing the total number of unique literal operands. If the literal operands relating to stack or branch operations can be minimized, significant operation pattern reduction can be achieved, leading to VLIW pattern reduction. VLIW pattern reduction can eventually lead to improved packing in the decoder memory. This forms the basis of the stack re-structuring idea, targeted on minimizing stack related literal operands, as described in Section 4.2.

Chapter 4

Methods for Achieving a Denser Decoder Memory

As described briefly in Section 2.1.2, the storage requirement for the VLIW instructions is provided by the two-level instruction memory system in UTDSP. An example of this memory system is shown in Figure 4.1.

In Figure 4.1 (a), a program consisting of two multi-op and one uni-op instructions is depicted. Figure 4.1 (b) then demonstrates how the instructions are stored in the two-level memory system. First, instructions 1 and 3 are packed and stored in the decoder memory at address 0x0. Second, two multi-op pointers, pointer 1 and pointer 2, containing the packing information for instructions 1 and 3, are stored at address 0x0 and 0x8 of the instruction memory. Instruction 2 is a uni-op instruction, so it stores its only operation at addresses 0x4 of the instruction memory. Pointer 1 stores two bit masks, 100 and 0100, and two decoder memory addresses, 0x0 and 0x0, one for each cluster. Finally, Figure 4.1 (c) shows how the original instructions can be retrieved. The bit masks of pointer 1 are used to select decoder banks 0 and 4 and to obtain the operations at address 0x0 of each bank. Hence, the original instruction 1 can be easily retrieved. Instruction 3 can be retrieved from pointer 2 in a similar fashion.

As was shown in Figure 4.1 (b), instructions 1 and 3 are easily packed into one decoder word at address 0x0, which is a good example of decoder memory usage. However, since every multi-op instruction in the decoder memory requires a corresponding multi-op pointer, the savings are somewhat reduced. Even so, M. Saghir[1] reported that the total memory required (instruction memory + decoder memory) is only 40–50% of the memory required when the instructions are stored in their unpacked format. This shows the effectiveness of the two-level memory system.

This chapter further explores ways to make the multi-op instructions more "packable", resulting in even smaller decoder memory requirements. The results show that some of the techniques are able to reduce the the decoder memory size by approximately 10% over that achieved by M. Saghir. There are four sections in total. Section 4.1 explores ways to improve on the previously proposed packing techniques that were briefly described in Section 2.1.3. Section 4.2 proposes a method to rearrange the stack so as to minimize the total number of unique constants. A method for minimizing the size of the stack is also proposed. Section 4.3 explores ways to perform register allocation such that unique operation patterns can be minimized. Finally, Section 4.4 summarizes the chapter.

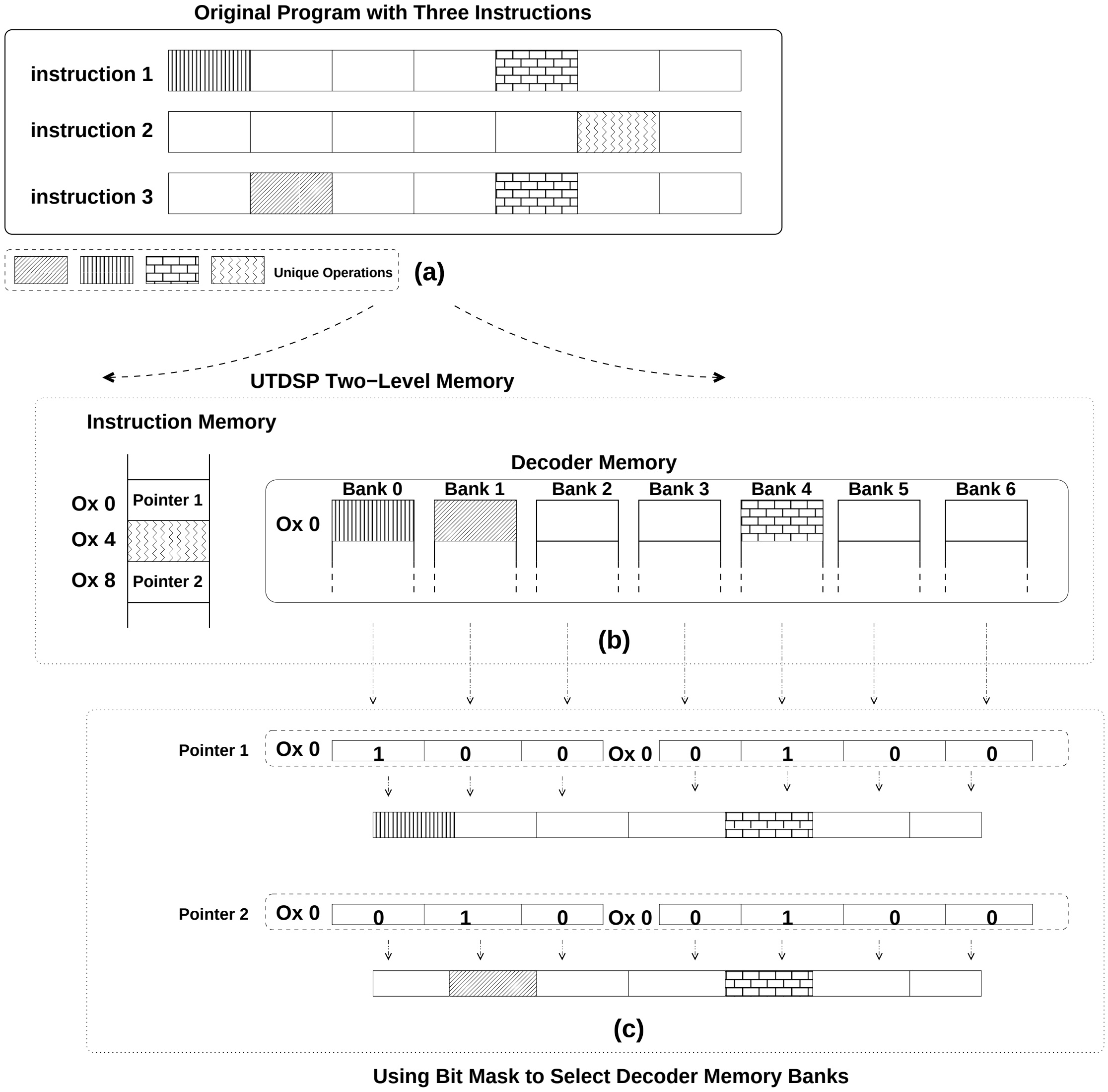

Figure 4.1: (a) Instructions in their original format. (b) Storing the instructions in the two-level memory system. (c) Using bit mask to retrieve the original instructions.

4.1 Achieving Compaction via Different Packing Techniques

This section reviews the UTDSP subset packing algorithm, proposes ways to augment it, and explores different packing schemes. Results and analytical evaluation of the effectiveness of the proposed packing techniques are also provided. The purpose of the techniques proposed are to minimize the decoder memory size and increase the memory utilization.

4.1.1 UTDSP Subset Packing Algorithm

The baseline packer used in the thesis incorporates the techniques of the two-field clustering and subset packing scheme as proposed by M. Saghir and S. Peng [19], where a subset is defined as:

Definition of Subset

> A long instruction X is said to be a subset of a long instruction Y if and only if every operation in X can be found in the corresponding field in Y, or the corresponding field in Y is empty.

The actual packing algorithm can be found on page 27 of [19] and can be summarized as follows:

Algorithm

1. Rank each field according to the number of its corresponding operations that belong to the unpacked instructions. The field with the most number of NOOPs in the unpacked instructions is assigned the highest rank.

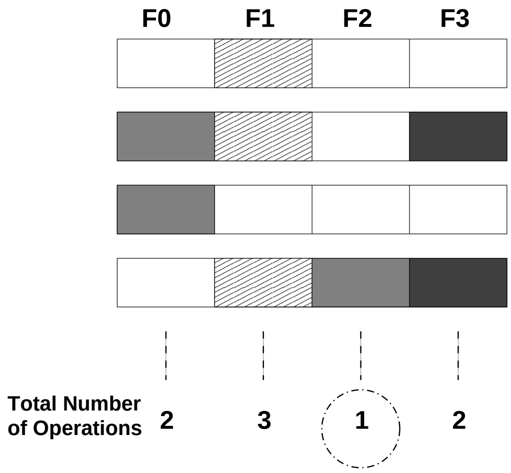

Figure 4.2: The field with the least number of operations is assigned the highest rank.

In Figure 4.2, the highest rank is assigned to field F2. The field with the highest rank, e.g. F2, is also called the target field.

2. Assign each unpacked instruction a cost according to the ranking of its fields that contain operations. A field that contains an operation is called an active field. The cost is calculated as follows:

$$Cost = \sum_{active fields} 2^{R_f} \times U_f \qquad \text{where,}$$

$$R_f = \text{rank of active field f}$$

$$U_f = \begin{cases} 1 & \text{if instruction uses an operation associated with active field f,} \\ 0 & \text{otherwise.} \end{cases}$$

3. Divide the unpacked instructions into two lists: a candidate list and a reserved list. A candidate list is a list containing instructions that have member operations associated with the target field. A reserved list is a list containing instructions that do not have member operations associated with the target field.

4. Remove a highest cost instruction from the candidate list and pack it into the decoder memory. If it is a subset of an existing word in the decoder memory, pack it into the existing word. If not, place it in a new word and search through the reserved list for instructions with the highest costs that can be packed into this decoder memory word.

5. Repeat step 4 until all instructions in the candidate list are processed.

6. Perform steps 1 to 5 again until all active fields have been processed.

7. Repeat steps 1 to 6 for instructions belonging to the other cluster.

4.1.2 UTDSP Subset Packing Algorithm with Binding-shifting and Commutative Transform

The original subset definition as used in step 4 of the packing algorithm determines whether two instructions can be packed into one memory word. Two enhanced subset definitions are proposed and used in place of the original subset definition in the algorithm. They are the *augmented subset* and *augmented commutative subset* definitions. Both definitions are based on the following idea.

If an instruction X, which is originally not a subset of instruction Y, can form an equivalent instruction that is a subset of instruction Y, then X can be replaced by its equivalent instruction and be packed into the same word as Y. The meaning of an equivalent instruction is one that executes and produces the same output as the original instruction.

Binding-shifting and commutative transform are two methods for generating equivalent instructions, leading to increasing the possibility of packing two instructions into one memory word. The definitions of the two methods and the enhanced subset definitions are provided below.

Definition of Binding-shifting

If N functional units of type U are present on a VLIW, an operation of type U has N possible bindings. If only one operation of type U is present in an instruction, N-1 equivalent instructions can be formed by changing the binding of this operation to the other N-1 positions.

Definition of Augmented Subset

A long instruction X is said to be an augmented subset of a long instruction Y if X or any of X's equivalent instructions can be a subset of the long instruction Y.

A VLIW processor often has multiple functional units of the same type, and the functional units have the same access to other hardware resources, such as the register files and the bus. Hence, the equivalent instructions formed as a result of binding-shifting are semantically identical to the original instruction.

Definition of Commutative Transform

An operation is commutative iff:

op (A, B) == op (B, A)

An operation is commutatively transformed if it is replaced with its commutative equivalence. An instruction is commutatively transformed if one or more of its operations have been commutatively transformed.

A notation used for indicating commutatively transformed instructions is: CF(original instruction)

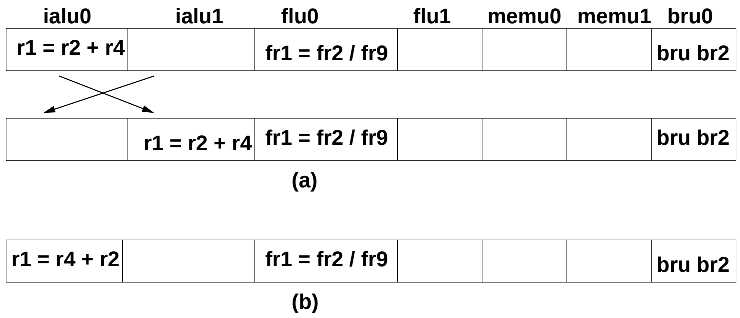

Figure 4.3: (a) Using binding-shifting. (b) Using commutative transform.

Definition of Augmented Commutative Subset

> A long instruction X is said to be an augmented commutative subset of a long instruction Y, if X, CF(X), any of X's equivalent instructions, or CF(any of X's equivalent instructions) can be a subset of the long instruction Y.

An example of an equivalent instruction generated by applying bind-shifting to an instruction is shown in Figure 4.3 (a). An equivalent instruction generated through commutative transform is shown in Figure 4.3 (b).

4.1.3 Packing Through Clique Partitioning

An idea to explore the possibility of finding a minimum decoder memory size is to map the instruction packing problem into the well-known clique partitioning problem [20]. The following mapping is used. An instruction is represented by a node. If one instruction is the subset of another instruction, then the nodes representing the instructions are connected by an edge, indicating that the two instructions can be packed into one decoder memory

```
CLIQUE_PARTITION( G(V,E) ) {
        P = nil;  // initialize partition
        while( G(V,E) not empty ) do {
                // compute maximum clique C subset of V in G
                C = MAX_CLIQUE( G(V,E) );
                P = P union {C};
                Delete C from G(V,E);
        }
}

MAX_CLIQUE( G(V,E) ) {
        C = {vertex with largest degree in G(V,E) };
        repeat {
                repeat {
                        U = { v in V : v not in C and adjacent to all vertices of C };
                        if( U is empty )
                                return C;
                        else  {
                                Select vertex v in U;
                                C = C union {v};
                        }
                }
        }
}
```

Figure 4.4: Clique partitioning algorithm.

word. If three nodes are connected to form a clique, then the three instructions represented by the nodes can be packed into one decoder word. A simple argument for the validity of this mapping can be found in Appendix A.1.

After the instructions are mapped onto a graph, the partitioning algorithm, shown in Figure 4.4, is used [21].

4.1.4 Divide and Conquer - Packing By Procedures

Another method used is to apply the UTDSP subset packing algorithm on a procedural basis. That is, instead of packing the entire program all at once, individual procedures are packed first. Then the packed results are merged into one final packing. The main reason for implementing this approach is for speed and efficiency. That is, if knowing that a reduction in packing a procedure can lead to a global reduction of packing every procedure, experiments can then focus on improving the packing of one function, leading to a faster obtainment of the results.

4.1.5 Results and Analysis

Benchmark	compress	li	m88ksim	go	perl	vortex
Total Decoder Memory Words	4604	11816	31164	32651	60256	58613
Decoder Memory Utilization %	98.22	98.34	98.93	99.49	98.84	99.79

Table 4.1: Subset Packing

Tables 4.1 to 4.5 each show the total number of decoder memory words obtained from using various packing methods on the different benchmarks. The unit of a decoder memory word is four bytes. Two equations are used to calculate the decoder memory utilizations and the percent improvement over the baseline packer. The equations are presented below.

Benchmark	compress	li	m88ksim	go	perl	vortex
Total Decoder Memory Words	4473	11434	30397	32049	58620	57967
Decoder Memory Utilization %	100	100	100	100	100	100
% Improvement over Subset Packing	2.8	3.2	2.5	1.8	2.7	1.1

Table 4.2: Augmented subset packing using binding-shifting.

Benchmark	compress	li	m88ksim	go	perl	vortex
Total Decoder Memory Words	4472	11433	30366	32015	58582	57926
Decoder Memory Utilization %	100	100	100	100	100	100
% Improvement over Subset Packing	2.9	3.2	2.6	1.9	2.8	1.2

Table 4.3: Augmented commutative subset packing using binding-shifting and commutative transform.

$$\text{Decoder Memory Utilization} = \frac{\text{Total Empty Words}}{\text{Total Decoder Memory Words}} \times 100, \\ \text{where word} = 4 \text{ bytes}$$

$$\text{The \% Improvement over Subset Packing} = \\ \frac{\text{Total Decoder Memory Words using Subset Packing - using Other Techniques}}{\text{Total Decoder Memory Words using Subset Packing}} \times 100$$

Table 4.1 shows the total number of words and utilizations of the decoder memory achieved by the baseline packer using the subset packing algorithm.

Benchmark	compress	li	m88ksim	go	perl	vortex
Total Decoder Memory Words	7437	17050	49283	35599	93681	89744
Decoder Memory Utilization %	61.38	68.97	63.31	91.81	64.44	65.57

Table 4.4: By Region

Benchmark	compress	li	m88ksim	go	perl	vortex
Total Decoder Memory Words	7458	17051	49194	-	-	-
Decoder Memory Utilization %	60.67	68.11	62.67	-	-	-

Table 4.5: Clique Partitioning

In Table 4.2, the effects of binding-shifting, applied to operations of integer and memory units, are shown. It cannot be applied to operations of floating point units because field clustering, which divides an instruction into two, as illustrated by Figure 3.1, forces operations of the two floating point units to be considered separately. Table 4.3 presents the data obtained when commutative transforming is applied on selective integer operations, namely, the ADD, AND, and OR operations. Tables 4.4 and 4.5 present data obtained by packing on a procedural basis and clique partitioning, respectively. Each table also shows the amount of decoder memory utilized by each packing method for each benchmark and the percent improvement over subset packing.

As can be seen from Table 4.1, the baseline packer is capable of achieving an average of 98.9% decoder memory utilization, hence demonstrating that the subset packing algorithm is effective in achieving dense packings. The time complexity of the algorithm is $O(n^3)$ [19]. By applying the algorithm using augmented subset and augmented commutative subset definitions, 100% decoder memory utilization is achieved, as shown in Tables 4.2 and 4.3. Binding-shifting is capable of eliminating empty words, or bubbles, in the decoder memory, by attempting to find an equivalent instruction that can be packed into an existing decoder word, when the instruction itself cannot. Therefore, by eliminating empty words, the utilization is able to reach 100%. The bubbles represent the NOOPs in the instructions. Since commutative transforms are only applied to three integer alu operations, the improvement in terms of total decoder memory size is incremental (i.e., small) over that achieved by binding-shifting. More detailed tables, describing the sizes of each individual decoder memory bank, can be found in Appendix A.2. Comparing the sizes of decoder memory banks for integer alu0 and alu1 in Table A.2 and A.3, size reductions as a result of commutative transformation can be observed.

When evaluating the effect of performing binding-shifting and commutative transform on the running time of the algorithm, it is clear that there is no significant overhead as both operations require only additional if statements in step 4 of the algorithm. Hence, the time complexity of the algorithm remains $O(n^3)$.

Both packing by procedures and clique partitioning give poor decoder memory utilization averages of 69% and 63.8%, respectively. The main problem with packing by procedures is that it does not have a global view for identifying instructions with higher ranking fields, that is, instructions possessing floating point operations. In the specInt benchmarks, floating point

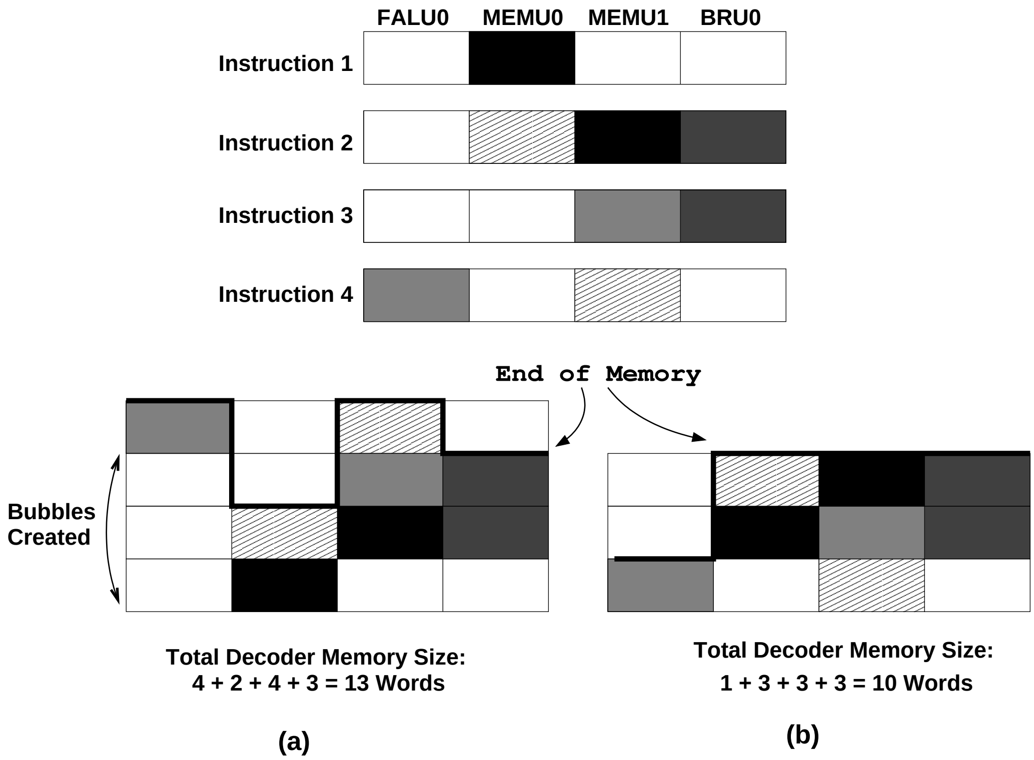

Figure 4.5: (a) Instructions are packed in sequential order. (b) Instructions with the highest costs are packed first.

operations are a small fraction number of all operations. The number of bubbles in the decoder memory can increase dramatically as a result of one instruction with a floating operation that is not processed early. In Figure 4.5 (a), the result of processing instruction 4 last creates three empty words in the first bank of the decoder memory, whereas processing it first eliminates the bubbles as shown in (b). The end of memory line indicates the highest address required for each bank and the total decoder memory size is computed by summing the sizes of each bank.

A similar problem exists with clique partitioning, that is, it lacks the ability to identify instructions with higher ranking fields. Even though the algorithm strives to identify and pack instructions that form the largest cliques initially, failing to identify even one instruction with higher ranking fields can lead to a significant increase in bubbles. Construction of the graph shows a time complexity of $O(n^2)$, and the partitioning of the graph to find an optimal partition, or minimal number of decoder words, is an NP-hard problem. Given the large number of nodes, and the large number of vertices connecting each node, clique partitioning becomes infeasible. Table 4.5 presents results for benchmarks compress, li and m88ksim. For benchmarks larger than m88ksim, the clique partitioning program was not able to terminate within a reasonable length of time, hence the results are not included.

4.1.6 Summary

The goal of this section is to explore methods for minimizing the decoder memory size and achieving a higher memory utilization. Various compaction techniques are examined and it can be concluded that the techniques of binding-shifting and commutative transform are effective and together, they achieve 100% in decoder memory utilization and up to 3.2% reduction in size, as shown in Table 4.3. Clique partitioning and packing by procedures on the other hand, are not effective in achieving higher utilizations or lower memory sizes as compared to the baseline packer. For future work, efforts should be placed on commutatively transforming more operations and creating clusters that can contain different fields. An example of this would be that the first cluster would contain both the integer units as well as the two floating units, hence allowing binding-shifting to take place for floating point operations.

4.2 Compaction Achieved via Stack Optimization

This section explores two methods to optimize the stack generated by the Trimaran compiler. The main purpose for optimization is to minimize the number of unique literal operands associated with stack-related memory operations.

First, locations for saving and restoring registers at call boundaries are reserved at addresses that are at fixed offset distances to the stack pointer (SP). Thus, the register saving and restoring operations in every function's prolog, epilog and call sites would consistently access locations at the same offsets relative to the SP. Hence, the constant operands of these memory operations are the same across every function if the same registers are to be saved or restored, making these operations much more compressible.

Second, the size of the stack is minimized through the following determination: if a local variable can be kept in a register, that is, its live-range is assigned a color, then it is not assigned a home location on the stack. If there are two local variables and both require home locations when neither can be allocated to a register, then the two variables may share the same home location on the stack if their live-ranges do not interfere. Through minimizing the stack size, hence the number of stack-related constants, the number of unique type II operations can be reduced.

The applicability of the optimizations to stacks maintained by other compiler systems is demonstrated. Furthermore, reducing the size of the stack is an added advantage to the main purpose of reducing stack-related constants.

4.2.1 Related Work in Stack Optimization

A recent work by J. Burlin [22] focuses on a method that assigns a home location to every local variable in the program. The goal is to generate an efficient assignment that minimizes the size of the stack. Burlin concludes that the best results can be achieved when an interference graph is built and variables, whose live-ranges do not interfere, are greedily assigned to the same location on the stack. Since the work is carried out in isolation from register allocation, it ignores the fact that it is unnecessary to assign home locations to local variables that are allocated to registers. This disregard for register allocation gives rise to the uniqueness of the approach proposed in this thesis, where stack assignments are generated during the process of register allocation and only unallocated variables are assigned home locations.

4.2.2 Trimaran's Original Stack Structure

Trimaran's stack structure, which resembles the stack structure of the HP Precision Architecture, is presented in Figure 4.6 (a). It is organized into three areas: the outgoing parameters, the local variables, and the register save areas. As the stack grows downwards, the outgoing parameters, placed on the stack by the caller function, are received as the incoming parameters by the callee function.

The *local variables* area stores local variables that are either arrays or structs. These variables are not subject to register allocation by the Trimaran

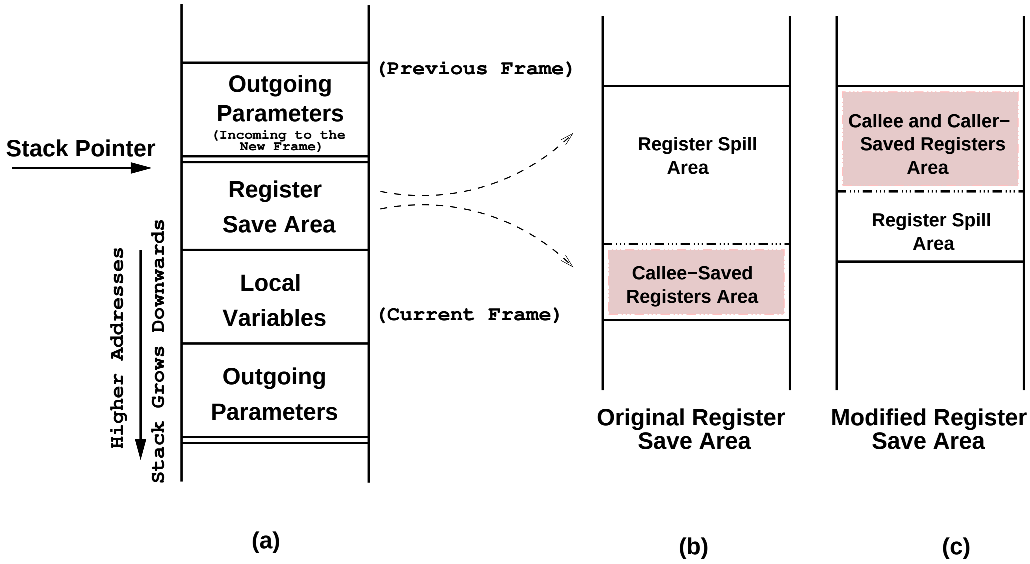

Figure 4.6: (a) Trimaran stack organization. (b) The original register save area. (c) The modified register save area.

compiler. Accessing a struct or an array variable requires an arithmetic operation to compute its address on the local variables area of the stack, and a memory operation.

An enlarged diagram of the *register save* area is provided in Figure 4.6 (b). The original register save area consists of two parts: the register spill area and the callee-saved registers area. The *register spill area* is an area reserved for spilling purposes. If a variable whose live-range cannot be allocated to a register, then it is spilled, that is, the variable is placed into the register spill area. Prior to the allocation process, every local variable subjected to allocation is assigned a home location on the register spill area. The exact location of the spilled variable in the register spill area is already predetermined. Therefore, if the variable is spilled during the allocation process, the allocator already knows its home location. The *callee-saved registers* area is an area where contents of the callee-saved registers are kept. The memory operations that save and restore the values of these registers are found in function prologs and epilogs.

However, there is no designated area for holding the values of the caller-saved registers. If the live-range of a variable extends over a procedure call and is allocated to a caller-saved register, the content of the register is saved to the variable's home location in the register spill area. As a result, contents of caller-saved registers are saved at locations scattered across the register spill area as the home location of each variable varies. The memory operations that save and restore values of caller-saved registers can often be located just before and after a procedure call.

There are a few problems associated with the currently defined register save area. Firstly, there is no designated storage area for the caller-saved registers. Secondly, the offset values used in the callee-saved registers area vary between frames. Finally, the register spill area is not efficiently utilized. These problems are further described below.

The offset distance, from the first address in the callee-saved registers area to the SP, can vary from frame to frame, giving rise to unique constants. The values of the offset distance is determined by the size of the register spill area. This is a compiler-specific problem and an illustration of the problem follows. Here, the code required for saving the content of register 4 to memory is shown.

Temp_reg = SP + #offset

(Temp_reg) = STORE (register 4)

If the spill area in the example is large, a large #offset value is required to skip over the area, setting Temp_reg to point to the appropriate location in the callee-saved registers area.

The last problem with the register save area is that the register spill area is not efficiently utilized. As discussed previously, every local variable subject to allocation is given a home location on the register spill area. However, only a few variables actually need to use their home locations. Thus, extra memory space is reserved without being used.

These problems have led to an inefficient stack structure. Thus, in the following two sections, two optimization methods are proposed that are also applicable to other compilers.

4.2.3 Restructuring of Register Save Area

As illustrated in Figure 4.6 (c), the register save area is restructured. Two changes are applied. Firstly, the register spill area switched locations with the callee-saved registers area, so that it is now adjacent to the local variables area. Secondly, the callee-saved area is expanded into the callee and caller-saved registers area, resulting in designated locations to save the contents of the caller-saved registers. Therefore, a caller-saved register, like a callee-saved register, will always be saved to a location that is at a fixed offset distance to the stack pointer, regardless of which variable it is allocated to.

Register saving and restoring code generated by Trimaran is provided in Figure 4.7. Operation [1] adds the appropriate offset to the stack pointer. The resulting address is stored in register Temp_reg and points to the designated stack location for register1. Operation [2] stores the content of register1 to the designated location. As every register has a designated location on the stack, #offset2 is a designated offset for register2. Similarly, operations [5] and [7] compute the addresses for the designated locations for registers 1 and 2 during register restores. Operations [6] and [8] restore registers 1 and 2 to their old values.

The above changes demonstrate the first steps toward stack structure optimization. Now there is a fixed offset value for every register and the caller-saved registers are no longer saved at locations scattered across the register spill area.

```
Temp_reg = SP + #offset1          [1]
(Temp_reg) = Store register1      [2]
Temp_reg = SP + #offset2          [3]
(Temp_reg) = Store register2      [4]
  •
  •
  •
```

(a)

```
Temp_reg = SP + #offset1          [5]
register1 = Load (Temp_reg)       [6]
Temp_reg = SP + #offset2          [7]
register2 = Load (Temp_reg)       [8]
  •
  •
  •
```

(b)

Figure 4.7: (a) Register Saving Code. (b) Register Restoring Code.

4.2.4 Register Spill Area Optimization

This optimization method targets the minimization of the size of the register spill area, hence reducing the possible unique constants in the spill code. The optimization is based on the idea that only variables whose live-ranges have been spilled require home locations. Thus, the assignments of home locations for spilled variables are done during the register allocation process. So, at the start of register allocation, the size of the register spill area is 0. As the coloring process continues and live-ranges begin to be spilled as a result of shortage of machine registers, the register spill area grows. Furthermore, if two variables, whose live-ranges do not interfere, are both spilled, then the same home location can be assigned to both variables. In other words, the two variables share the same memory location, which is also referred to as spill sharing.

This optimization is similar to the approach taken by J. Burlin, where two non-interfering live-ranges may share the same memory location. However, in Burlin's approach, every local variable is assigned a home location regardless of whether it will be spilled or not since the assignment is done prior to and in isolation from register allocation. But in the approach taken here, home locations are only assigned to spilled variables during the process of register allocation. Thus the resulting stack size is much smaller, providing an unique optimization method.

4.2.5 Implementing Stack Optimization for Other Compilers

The purpose of this section is to illustrate that the optimization methods proposed in Sections 4.2.3 and 4.2.4 are applicable to stack layouts of other compilers. The other compiler examined in this section is the SUIF1 compiler [23]. Although the restructuring of the register save area is not directly applicable to SUIF1, the concept of reordering areas to minimize the required constants is still novel and applicable to SUIF1.

SUIF1 is taken under study since it was the compiler used in the previous UTDSP project. Its stack organization is presented in Figure 4.8 (a). The stack structure is similar to the Trimaran's structure in that it is also divided into three areas: the outgoing parameters, the local variables and the register save areas.

The register save area is strictly used for saving and restoring the contents of registers. Since all registers are treated as callee-saved registers by the allocator, the register saving code can only be found in function prologs and epilogs. The register save area does not include a register spill area, as the allocator does not spill registers. The allocator either keeps a variable in a register, or it keeps it in memory on the local variables area of the stack.

Sample code for saving and restoring registers is provided in Figures 4.8 (b) and (c), respectively. The immediate operand of RegSave_top is indicated by the dashed arrow in Figure 4.8 (a). Operation [1] initializes a register with the appropriate offset and operation [2] adds the offset to the stack pointer, a63. The resulting address points to the top of the register save area. This address is stored in register a1 which is used as a pointer. Operation [3]

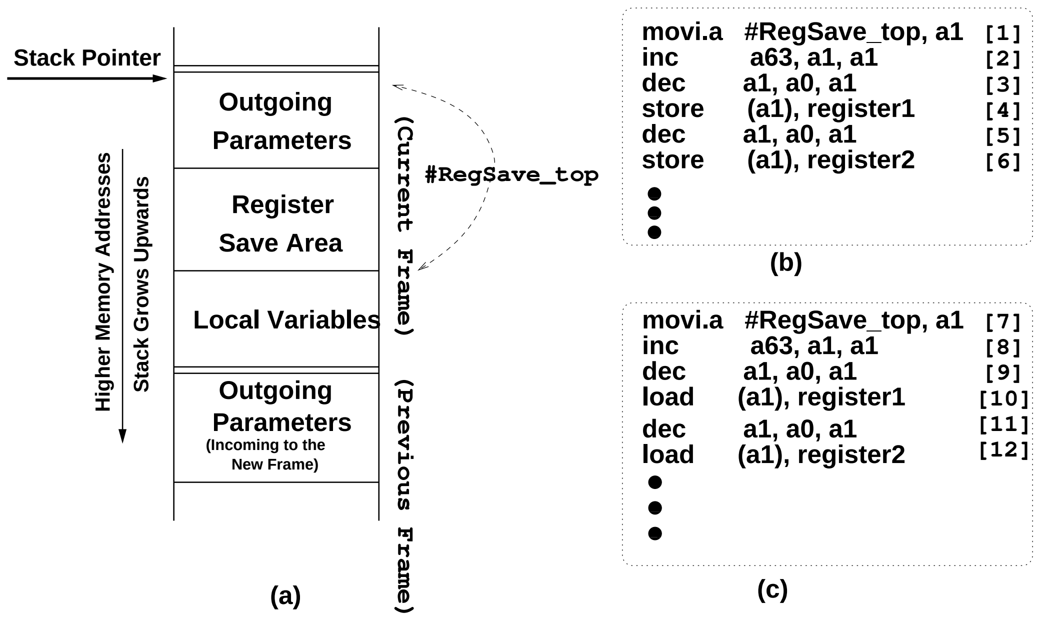

Figure 4.8: (a) SUIF Stack Organization. (b) Register Saving Code. (c) Register Restoring Code.

decrements the pointer - a0 is hardwired to 4 - and operation [4] stores a register at the corresponding location on the register save area of the stack. For every register stored on the stack, a similar pair of operations is generated. Operations [5] and [6] are an example of such a pair. Operations [7] - [12] are similar to operations [1] - [6] but they perform the reverse function, where the contents of the registers are restored to the old values.

The optimization method proposed in Section 4.2.3 is not applicable to the SUIF stack for the following two reasons. First, the register save area does not subdivide into smaller areas. Second, as can be observed from Figures 4.8 (b) and (c), only one literal operand is required throughout operations [1] - [12], that being #RegSave_top. Subsequent memory locations can be computed using a decrementing operation, such as in operation [3]. These operations are unlike the register saving code generated by Trimaran, as illustrated in Figure 4.7, where the address of each location is computed from an offset. However, the need for the #RegSave_top offset can be eliminated if the outgoing parameters and the register save areas are swapped. Hence, the concept of area reordering would still be applicable.

The second optimization method, however, can be applied to SUIF's register allocator so the local variables area can be minimized. As discussed above, the allocator of SUIF either keeps a variable in a register or it keeps it in memory on the local variables area of the stack. Even though some local variables may be allocated to a register, they are still given home locations on the local variables area of the stack. This results in poor stack utilization. Thus, using the optimization method described in Section 4.2.4, the SUIF stack can be efficiently utilized to only hold variables that are not allocated to registers.

4.2.6 Results and Analysis

Results are collected and used to evaluate the effectiveness of the proposed optimization methods. The baseline packer is used to obtain the sizes of the decoder memory.

benchmark	compress	li	m88ksim	go	perl	vortex
(a) Original Stack Organization						
Total Decoder Memory Words	4604	11816	31164	32651	60256	58613
(b) Restructuring of Register Save Area and Spill Area Optimization without Spill Sharing						
Total Decoder Memory Words	4542	10446	28774	29705	54196	48833
% Improvement over Original Stack Organization	1.3	11.6	7.7	9.0	10.1	16.7
(c) Restructuring of Register Save Area and Spill Area Optimization with Spill Sharing						
Total Decoder Memory Words	4536	10430	28724	29705	54101	48737
% Improvement over Original Stack Organization	1.5	11.7	7.8	9.0	10.2	16.8

From Table 4.2.6, it can be seen that the optimization methods of register save area restructuring and spill area optimization with or without spill sharing are effective techniques capable of achieving up to 16.8% of decoder memory size reduction in benchmark vortex. The benchmark compress is a

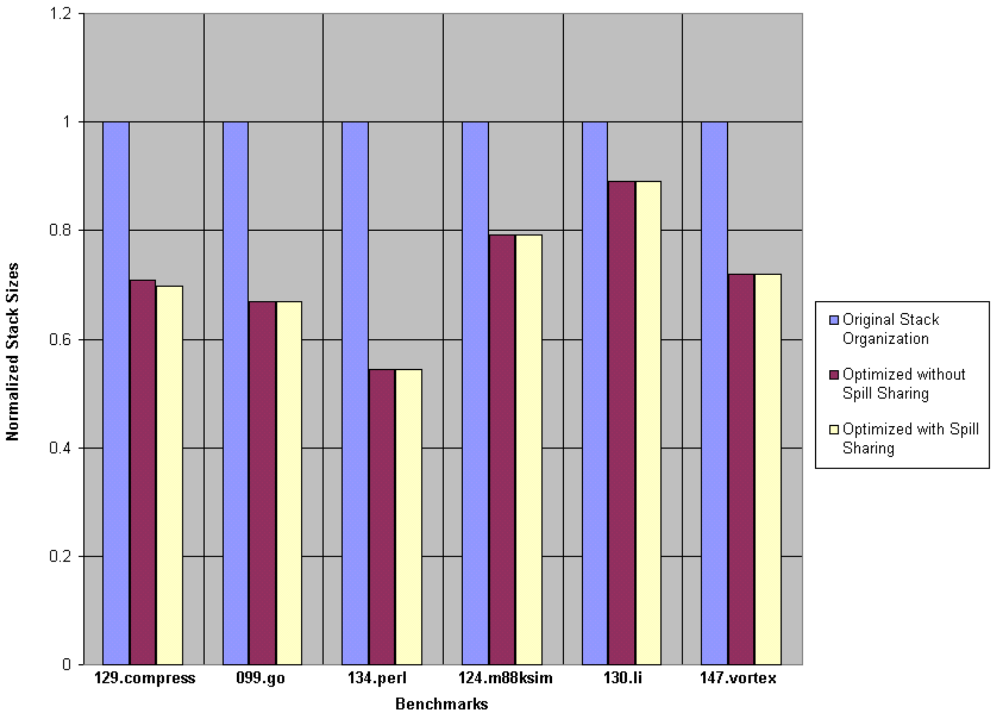

Figure 4.9: Normalized Stack Sizes.

relatively smaller benchmark and has significantly less stack-related constant operands. Hence, it benefits much less from the optimizations than the larger benchmarks.

Figure 4.9 shows the stack sizes of the benchmarks for the optimization methods mentioned above. In each case, the stack size is normalized to the upper bound, that is, the stack size obtained with the original stack structure. The technique of spill sharing does not contribute to major reduction of stack sizes, after both optimization methods have been applied. This directly corresponds to the insignificant decoder memory size reduction shown in

Table 4.2.6 (c).

4.2.7 Summary

Section 4.2 introduces two optimization methods capable of reducing the number of stack-related constants. The first technique, register save area restructuring, allows registers to be saved at designated locations of the stack. As a result, the register saving operations appear repetitive, and hence more compressible. The second technique, register spill area optimization, reduces the size of the stack by assigning home locations to spilled variables only and allowing two spilled variables whose live-ranges do not interfere to share the same location. As the size of the stack is reduced, the number of stack-related constants is also reduced. Furthermore, this section illustrates the applicability of the proposed techniques to other compiler systems, namely SUIF1. The techniques are also shown to be effective, as up to 16.8% of decoder memory size reduction is achieved for Trimaran on UTDSP.

For future work, Trimaran's allocator can be modified to generate register saving code that uses decrement and increment operations in computing subsequent memory addresses. Potentially, more offset constants can be eliminated, leading to further improvement in packing. The restructuring of stack area and the spill techniques introduced has no effect on the execution performance of the program. The only trade-off is in the speed of the allocation process as it needs more time to check whether two live-ranges interfere or not and assign locations accordingly.

4.3 Compaction Achieved via Coloring

In Chapter 3, three different types of operations are introduced. It has been defined that type I operations are operations where all the operands are registers and type II operations are ones where at least one, but not all the operands are registers. However, as a result of inefficient register allocation, operation patterns are not minimized. To better illustrate an efficient process, an example is shown as follows. An operation pattern, e.g., $r1 = r2 + r3$, emerges during the process of allocation. The operands of a subsequently encountered operation, e.g., $r1 = vr3 + vr6$, should then ideally be assigned registers that would make the two operations identical. ($vr3$ and $vr6$ are virtual registers that must be allocated to physical registers.) In this case, $r2$ should then be assigned to $vr3$ and $r3$ should be assigned to $vr6$ if both registers are available.

Thus, the objective of this section is to explore three different ways of performing register allocation to minimize the resulting number of operation patterns. The first approach is to perform register allocation while maintaining a database of previously allocated patterns. This database is then consulted when making coloring decisions for subsequent live-ranges. The second approach is to experiment with the order in which live-ranges are colored. The last method that is analyzed is the incorporation of simulated annealing into the coloring process.

Register allocation has been extensively studied where heuristics are used to minimize the number of memory accesses by the allocated code. The heuristics derive priorities for live-ranges, which dictate the ordering in which they are considered and their assignment of colors. The heuristics guide the allocator to perform under various constraints. Any violation of the constraints will result in an excessive number of memory accesses, as shown in previous studies [9]. Thus, the main challenge facing the analysis is in finding ways to minimize the operation patterns and at the same time, obeying the constraints. Furthermore, the type of operations that can benefit from the optimizations proposed in this section is the type I operations. However, type I operations only account for a small fraction of the total operations. Hence, the reduction in operation patterns may be insignificant, leading to incremental improvement in packing. Though faced with these challenges, the idea of minimizing the operation patterns in register allocation is novel and worth exploring.

4.3.1 Coloring with a Database

One approach to minimizing operation patterns is to maintain a database of existing operation patterns. This database is consulted each time the allocator chooses a color for a live-range. Ideally, the goal is to select a color for a live-range that will make the operations spanned by the live-range identical to some existing operations. Hence, the total unique operation patterns can be minimized. As illustrated in Figure 4.10 (a), the live-range spans four operations. Figure 4.10 (b) indicates that there are three patterns in the database and five available registers for the live-range. A pattern-aware color selection process would then assign $r3$ to the live-range, making two operations, marked by asterisks in Figure 4.10 (b), identical to two patterns already in the database. As a result, only two new operation patterns are created. On the other hand, if $r1$ is selected, four new patterns would be

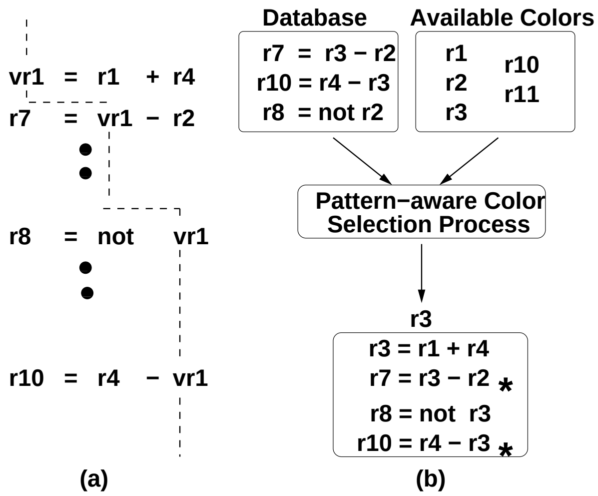

Figure 4.10: (a) A live-range spanning multiple operations. (b) r3 is selected, which allows operations, marked by asterisks, to be made identical to existing operations in the database.

created.

Original Color Selection Process

Prior to any coloring, the registers are divided into two pools: the caller-saved register pool and the callee-saved register pool. The allocator then attempts to minimize the call overhead by comparing the benefits of selecting certain registers from each pool. After it determines the most beneficial pool, it gathers all the available registers from that pool and passes them to the color selection process where one register is finally selected. The original color selection process is straightforward. It simply selects the register with the lowest register number. For example, in Figure 4.10 (b), r1 would be selected. As shown in the above example, it is evident that choosing the lowest register number does not result in the creation of the fewest number of operation patterns. Thus, a new color selection process, called the pattern-aware color selection process, is analyzed.

Pattern-aware Color Selection Process

As illustrated in Figure 4.10 (b), a pattern-aware color selection process requires a pattern database. The database provides a history of the previously encountered operation patterns, and thus influences a better selection process.

The Database itself is implemented as a hash table that stores the fully allocated type I and type II operation patterns so far. Its inputs and outputs are as follows. The input to the database is the environment of a nearly allocated operation. A nearly allocated operation is an operation where only one of its operands is still not allocated. If an operation spanned by a live-range is not a nearly allocated operation, it is not considered by the pattern-aware selection process. A literal operand is considered allocated. For example, operation $gpr1 = vr3 - 2$ would be considered as a nearly allocated operation where only the first source operand has not been allocated. If a nearly allocated operation is spanned by a live-range, then the operand that needs to be allocated will be the variable of the live-range. The environment of a nearly allocated operation is defined as a five-tuple consisting of an opcode and four operands. The five-tuple is denoted as follows: (opcode, dest_operand1, dest_operand2, src_operand1, src_operand2), where the unallocated operand is replaced by an asterisk (*) and the unused operand is replaced by a dash (-). For example, the environment of operation $gpr1 = vr3 - 2$ is represented as $(sub,\ gpr1,\ -\ ,\ *,\ 2)$. The database then searches for patterns that match the input environment, where the operand indicated by an asterisk is allowed to take on any value. For example, patterns such as (sub, $gpr1$, $-$, $gpr2$, 2) and (sub, $gpr1$, $-$, $gpr11$, 2) would both be identified as matches while (sub, $gpr2$, $-$, $gpr1$, 2) would not be a match.

The output from the database is a bit vector. The width of the bit vector equals the total number of machine registers. For example, the width of the vector returned for an integer operation query, as illustrated in the above example, would be 64 as there are 64 general purpose registers or integer registers. A 1 in bit position n of the vector indicates that register n has been used as an operand in the asterisk position of operations which qualify for the input environment. For example, the returned bit vector for the example query illustrated in the last paragraph would be a bit vector that has bits 2 and 11 as 1 and the others as 0.

The Pattern-aware Color Selection Process is shown in Figure 4.11. The algorithm used in this process is described as the *frequency-based color selection* algorithm. The algorithm requires four inputs: the live-range, a bit vector of available registers, a pattern database, and a list of opcodes sorted in decreasing order of frequency. The list of opcodes is generated by ranking the usage frequency of each opcode. A table describing the 14 most frequently used opcodes in decreasing order is provided in Figure 4.12. As indicated in the table, the opcode MOVE has the highest frequency.

Step 1 of the algorithm iterates over the list of opcodes sorted in decreasing order of frequency. The opcode being considered is then set as the current_opcode in step 2. Each operation spanned by the live-range is then

```
                Algorithm: Frequency−based Color Selection

Input: (1)  a live−range
       (2) a bit vector indicating the available colors
       (3) a database storing operation patterns
       (4) a list of opcodes sorted in decreasing frequency order
Output: a color (to be assigned to the input live−range)


Definitions:
(1)  A nearly allocated operation is an operation where only one of its
     operands has not been register allocated.
(2)  The environment of a nearly allocated operation is a 5−tuple indicated as:
      (opcode, dest_operand1, dest_operand2, src_operand1, src_operand2),
      where the unallocated operand is replaced by an asterisk (*).
(3)  The ENVIRONMENT function returns the environment of a nearly
      allocated operation.
(4)  The LOOK_UP function accepts an environment as input, queries
     the database, and returns a bit vector indicating colors that have been
     assigned to operands in the asterisk position of the environment.


1.  Loop over the list of opcodes
            2.  current_opcode <= opcode in the list
            3.  If (the live−range passes through a nearly allocated operation
                whose opcode equals current_opcode) then
                        4.  current_environment <=
                                   ENVIRONMENT(nearly allocated operation)
                        5.  used_colors <= LOOK_UP(current_environment)
                        6.  suggested_colors <= used_colors AND available_colors
                        7.   If (suggested_colors is zero) then
                                8.  continue to examine the next opcode in the list
                        9.   Else
                                10. return the position of the first 1 bit
                                    in suggested_colors
                             End If
           11.  Else
                        8.  continue to examine the next opcode in the list
                End If
  End Loop
12. return the position of the first 1 bit in available_colors
13. update the database
```

Figure 4.11: Frequency-based color selection algorithm.

decreasing order of frequency

MOVE
ADD_W
CMPP_W_GEQ_UN_UN
CMPP_W_LLEQ_UN_UN
SUB_W
AND_W
OR_W
SHL_W
SHR_W
ADDL_W
SHRA_W
CMPP_W_GT_UN_UN
CMPP_W_LT_UN_UN
DIV_W

Figure 4.12: Opcodes listed in decreasing order of frequency.

examined. If the operation spanned by the live-range is nearly allocated with the opcode being the same as the current_opcode (step 3), then the environment of the operation is extracted (step 4). Following this, the database is queried to find used registers. Step 6 examines whether there is a used register that is also an available register. If this type of register exists, the color is returned, else the search continues. If step 12 is reached, there are no used registers and any available color can be returned. Using the original color selection scheme, the register with the lowest register number is returned. If any nearly allocated operation spanned by the live-range becomes fully allocated as a result of coloring the live-range, the newly formed operation pattern needs to be inserted into the database. Thus, step 13 updates the database. As can be seen, the algorithm targets the minimization of the operation patterns with the most frequently appearing opcodes, hence it is called the *frequency-based color selection* algorithm.

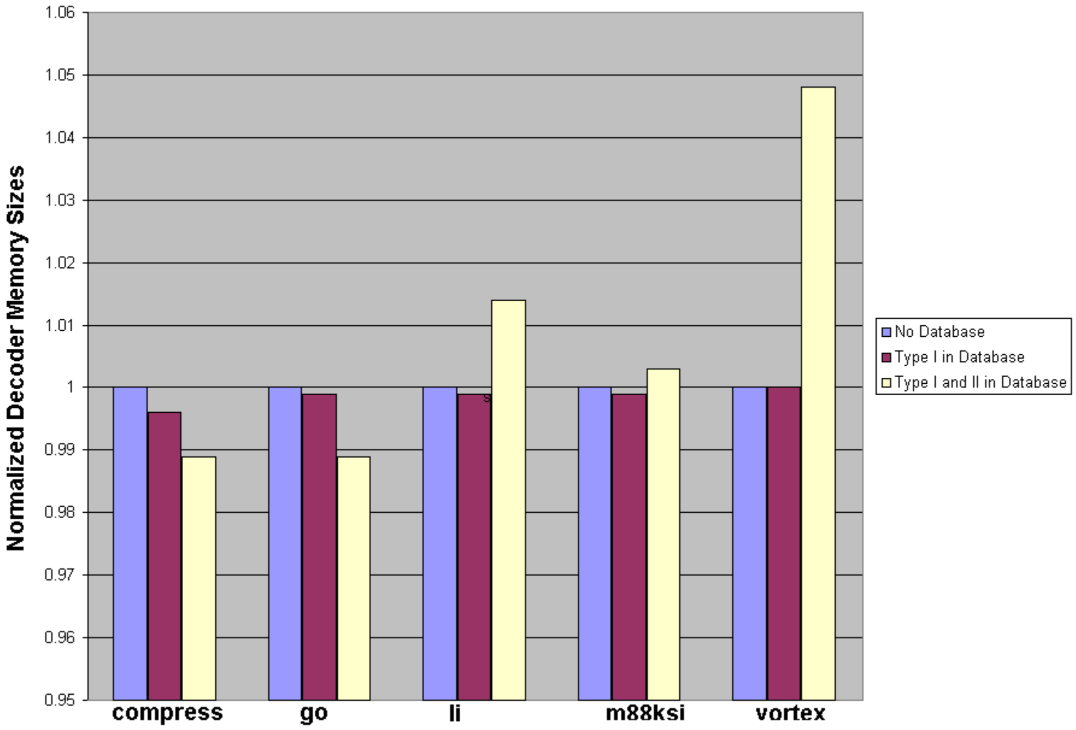

Figure 4.13: Normalized Decoder Memory Sizes for the Database Optimization technique.

Results and Discussion

When only type I operation patterns are stored in the database, the algorithm may only attempt to minimize type I operation patterns. When both type I and type II operation patterns are stored, the algorithm may then attempt to minimize both types of operation patterns. Figure 4.13 shows the storage requirements of the benchmarks for optimizing type I, and both type I and type II operation patterns using the database approach. The baseline packer is used to obtain the decoder memory sizes. In each case, the decoder memory size is normalized to the size obtained when coloring is performed without using a database. As shown in Figure 4.13, the reductions achieved by both optimizations are insignificant and in cases, worst than that achieved without using a database. There are two reasons that can explain the poor performance of this optimization technique. Firstly, the number of type I operations is a small fraction of the total number of operations. Hence, minimizing for type I patterns indicates insignificant reduction. Secondly, type II operations are difficult to be made identical because often, the distinguishing factor is in the constant operands of the operations, not in the register operands.

Furthermore, the frequency-based color selection algorithm can be defeated by scenarios such as the one illustrated in Figure 4.14. Figure 4.14 (a) shows a live-range that spans four operations, three of which are nearly allocated operations. Figure 4.14 (b) shows that the frequency-based color selection algorithm would select r3 because addition has a higher opcode frequency than both *and* and *not*. However, r4 would be a better choice in this scenario since two of the nearly allocated operations can be made identical to the existing patterns. However, the choice of r3 only leads to the creation of one identical operation. Future work should focus on improving the color-selection algorithm.

4.3.2 Live-range Coloring Order

The order of coloring is an important factor in register allocation. As the quality, in terms of the amount of spill, is determined largely by the order of coloring. Priority functions have been proposed by various researchers to

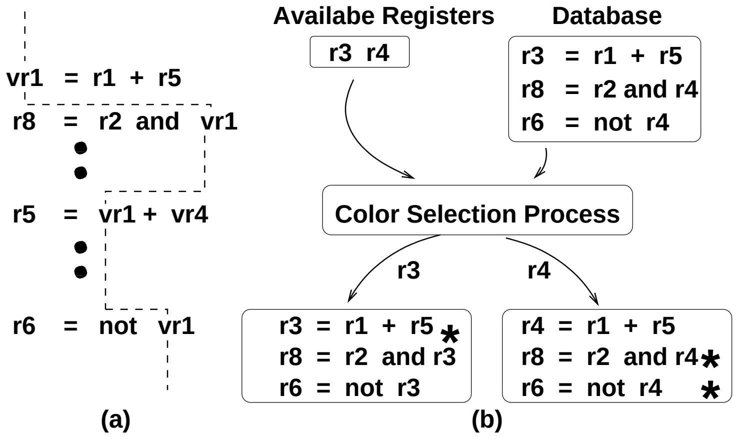

Figure 4.14: Scenarios that defeat the Color Selection Heuristic.

dictate the order in which live-ranges are selected [9]. In general, the priority function relates to the amount of spill code that can be saved by allocating a variable to a register as opposed to spilling the live-range of the variable [9].

Two simple experiments are conducted to observe the effect of live-range ordering on packing. The first experiment is to color the live-ranges with the least number of interference edges first. The second experiment is to color the live-ranges with the most number of interference edges first.

Results and Discussion

Figure 4.15 shows the decoder memory requirements for three coloring orders: highest priority first, least interference edges first, and most interference edges first. In each case, the decoder memory size is normalized to the size obtained when live-ranges with higher priorities are colored first. The benchmark compress is able to achieve a 9% size reduction when the live-ranges with the most number of interference edges are colored first. However, the

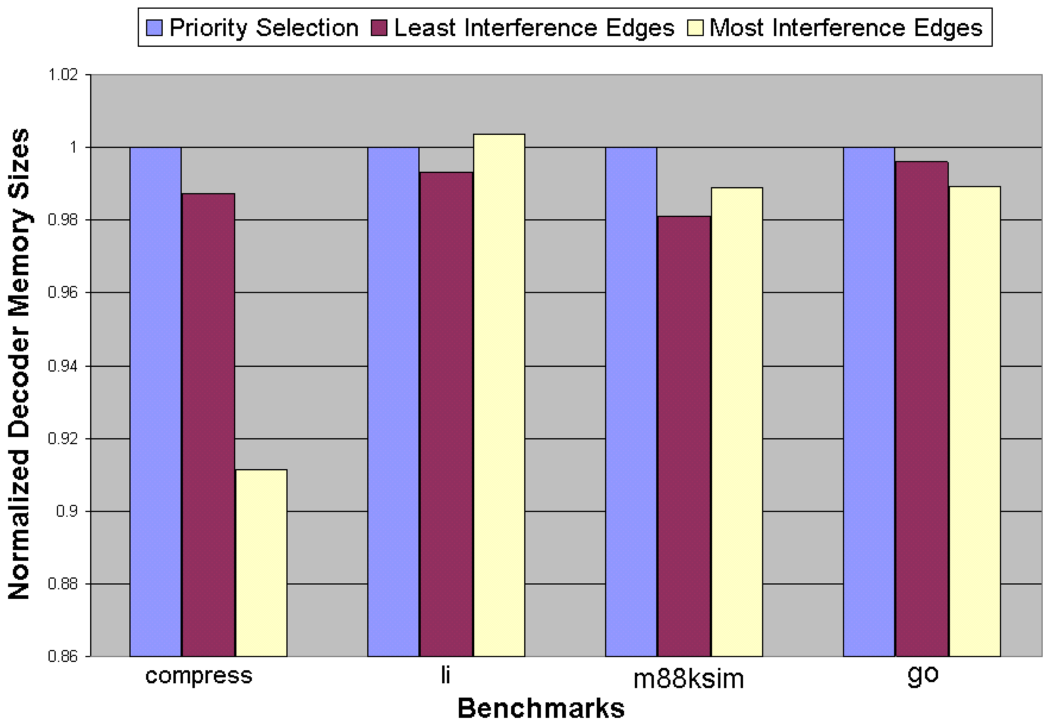

Figure 4.15: Normalized decoder memory size for different live-range coloring orders.

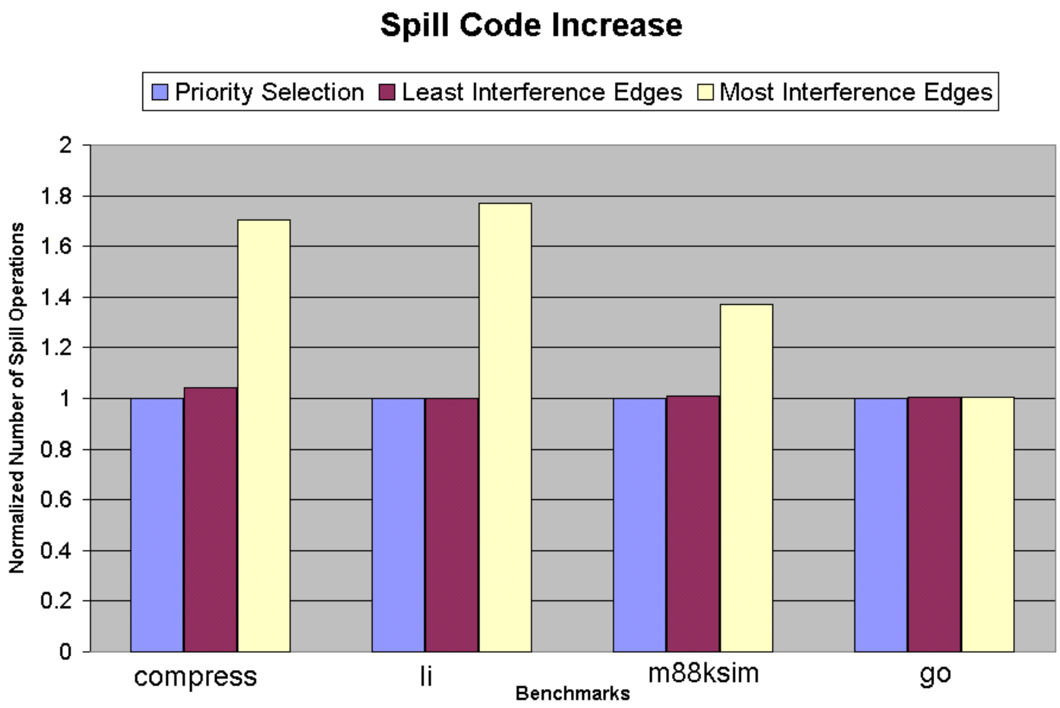

Figure 4.16: Spill code increase as a result of changing the order in which live-ranges are colored.

reduction is achieved at the expense of a 70% increase of spill code as shown in Figure 4.16. The benchmark m88ksim is able to achieve a 2% size reduction when live-ranges with the least interference edges are colored first. Again, the reduction is achieved at the expense of 1% spill code increase. The increase in spill code means a decrease in the execution speed.

Summary

The priority function implemented in the baseline allocator is capable of minimizing the spill code effectively. The results obtained from the two coloring orders tested in this section suggest that a trade-off between an increase in spill code and a decrease in decoder memory size is possible. However, as the spill code can significantly decrease the execution performance of a function, it is not wise to exploit such a trade-off.

4.3.3 Live-range Coloring with Simulated Annealing

This section explores ways to incorporate the general optimization technique of simulated annealing (SA) into register allocation. The goal is to find ways of performing allocation such that the unique type I and type II operation patterns can be minimized and ideally, without affecting the execution performance.

This section is divided into five subsections. The first subsection briefly describes approaches that were explored to incorporate SA into register allocation. The second subsection proposes a graphical formulation to represent a live-range and operations spanned by the live-range. The formulation further defines the term "degree" to be the number of operations that are not spanned by the live-range yet have identical patterns as the operations spanned by the live-range. Minimizing the degree of a live-range minimizes the total unique operation patterns. The third subsection proposes two techniques to recolor a live-range so that its degree may be minimized. The first recoloring technique is a greedy algorithm and the second technique incorporates SA into the recoloring process. The fourth subsection presents the results for the two recoloring techniques. Finally, the last subsection summarizes this section.

A Brief Overview of Approaches Attempted in Incorporating Simulated Annealing into the Allocator

Simulated annealing (SA) is a well-known optimization technique. It mimics the annealing process used to gradually cool molten metal to produce high-quality metal objects [24]. It is often used to search through a large solution space to find solutions with lower costs. It differs from a greedy algorithm because it has a probability of accepting a solution with a higher cost, or a worse solution.

Different approaches have been explored to integrate SA into live-range coloring. Two of these approaches are briefly discussed in this section. The first one is a simple approach and the second one is an improved version of the simple approach.

The simple approach first randomly colors every live-range, providing a coloring that can be used as the initial state for the annealing process. A new state is then generated by randomly selecting a fraction of the total number of live-ranges to recolor with new colors that are also chosen at random. After every live-range has received a color, the live-ranges that are assigned the same colors as their neighbors in the interference graph are spilled. The allocated code is then passed to the scheduler for long instruction scheduling and further to the packer where a cost, in terms of the total decoder memory size, is obtained.

The main problem with the simple approach is that it produces an unreasonable amount of spill code, which is on the order of four times more than the amount produced by the original allocator. Even though each new state has an excessive amount of spill code, the cost of each state can still progressively decrease. This is because the operations in the spill code are highly repetitive, hence they can be easily compacted. Eventually, the cost can be reduced to one that is 15 – 20% lower than the original cost achieved by the code that is allocated by the baseline compiler. In other words, the size of the decoder memory required for packing a function allocated by the simple annealing approach decreases by 15 – 20%. However, when all the functions are packed together, 20 – 25% more decoder memory is required. Thus, the simple approach is not an efficient method for incorporating SA into live-range coloring, and also seems counter-intuitive. The reason is as follows.

In the baseline allocator, two registers, register 1 and 2 of every file type, are designated as spill registers. When the live-range of a variable is spilled, subsequent referencing of the variable requires memory operations that load the contents of the variable into the spill registers. In Figure 4.17 (a), operations 1 and 2 cannot be made identical because the live-ranges of vr1 and vr5 interfere. However, if both live-ranges are spilled, then vr1 and vr5 in operations 1 and 2 can both be assigned to the same spill register, in this case, r1. The effects of spilling both live-ranges are illustrated in Figure 4.17 (b), as operations 1 and 2 become identical. At the same time, there is now a possibility that operations 3 and 4 can be made identical. Thus, at the expense of four pairs of spill operations, two operations can be made identical. Each pair of spill operations consists of an addition and a memory operations. Furthermore, since the spill operations are highly repetitive, the four pairs of spill operations generated in this case are likely to be packed with existing spill operations. Hence, the effect of spilling both live-ranges is the reduction of one operation pattern. As more live-ranges are spilled leading to more pattern reductions, the packing of a function also improves. However, when there are many procedures, the excessive amount of spill code begins to occupy more slots in the decoder memory, hindering other operations from being packed into existing memory words.

Another problem with the simple approach is that one cycle of the annealing process, from the allocator to the end of packer, requires a long time. For some functions, one to two minutes are required for each cycle. The purpose of SA is then defeated since the search space cannot be exhaustively explored given such a long turn-around time. Hence, an improved version of the simple approach is analyzed, which addresses the problems experienced by the simple approach.

To minimize the problems experienced by the simple approach, **an improved version of the simple approach** is studied. In the new approach, the annealing process begins with an initial state of valid coloring generated by the original allocator. The next state is then generated by randomly recoloring a fraction of the total live-ranges. If a live-range selected for recoloring is assigned a color that is the same as any of its neighbors, it is immedi-

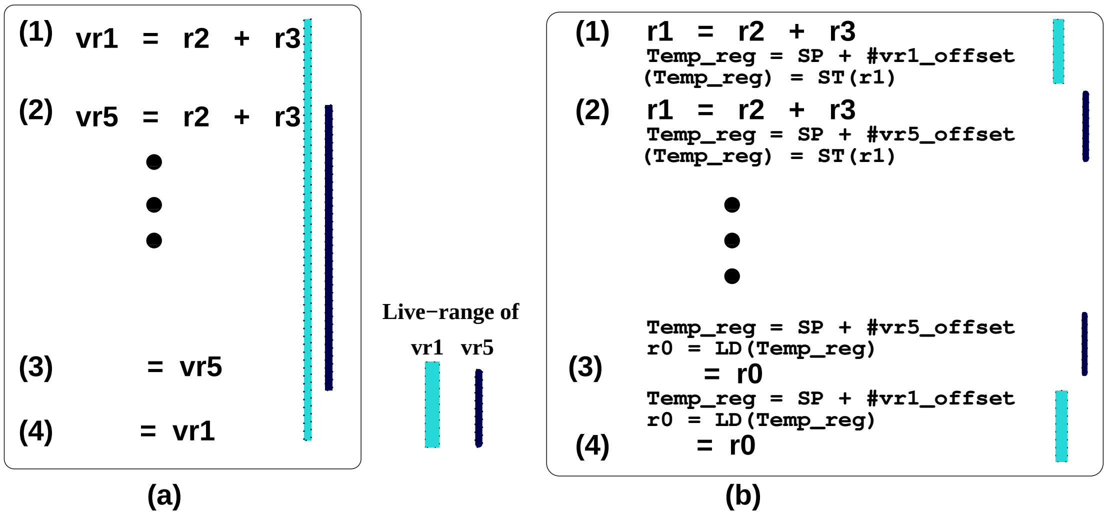

Figure 4.17: Spilling of two live-ranges to create identical operations.

ately recolored again until it is assigned a new color, not used by any of its neighbors. If it cannot be recolored due to a shortage of machine registers, then the live-range is spilled. A spilled live-range can also be recolored and assigned a register, if one becomes available during the recoloring process. The annealing process is also shortened in this revised approach by redefining the cost to be the sum of the spill cost, function call overhead and the total operation patterns in the allocated code. The cost can be efficiently computed for each state. Hence, the code with the new color assignment no longer needs to be scheduled and packed to obtain the decoder memory size as its cost. Using the new cost definition, each annealing cycle is shortened to 5 – 10 seconds.

Although the improved approach addresses the problems in the simple approach, it does not create a more efficient SA process. In the improved approach, a large part of the search space, where color assignments violate the interference constraints, is pruned out, thus speeding up the search process. However, the results are still far from promising as the spill code generated is still on the order of two to three times more than the original. As a result, the new costs for subsequent states as derived from the three parameters are always higher than the initial state.

There are quite a few problems associated with the improved approach. The main problem is that it lacks the ability to make two important coloring decisions. Firstly, when there is a shortage of registers, it is unable to select and spill the live-range that can lead to a minimal increase in spill code. Secondly, it is unable to distinguish between caller-saved and callee-saved registers. As a result, it cannot take advantage of the calling convention to minimize the register saving code. Another minor problem is that the improved approach does not encourage register reuse. For example, if both registers 3 and 31 are available, it is equally likely to choose either register whereas the baseline allocator would always choose an available register with the lowest register number. Reusing registers can also lead to a smaller number of unique operation patterns. The last problem, which is unavoidable, arises from its redefinition of the cost for a coloring state since the instruction scheduling aspect is not factored into the new cost definition. Thus, further approaches need to be analyzed to find the ultimate solution to the incorporation of SA into the coloring process, where all coloring constraints are obeyed. In the next section, four rules are presented which are used to model the constraints of an optimized register allocation.

The Graphical Formulation of a Live-range and the Operations Spanned by the Live-range

After analyzing the many problems described in the previous section, the final method that was tried for incorporating SA into coloring is derived. This method, or optimization, is called recoloring with SA optimization. The main basis for the optimization is that it obeys the following four coloring rules. Firstly, the benefits of assigning either a caller-saved or callee-saved register to a live-range are weighed and the decision made accordingly. Secondly, the interference constraint is obeyed. Thirdly, register reuse is encouraged, but not required. Finally, the live-range that is chosen to be spilled is made based on a prior analysis of the resultant spill code.

For the optimization method to function properly, it has its own techniques for formulating the live-range and the operations that it spans. As shown in Figure 4.18 (b), a large circle, which is referred to as a live-range boundary, encloses little bubbles. Each little bubble represents a unique operation pattern spanned by the live-range under consideration. The number inside the bubble indicates the total number of operations having the pattern represented by the bubble. As shown in Figure 4.18 (a), the live-range which is assigned register r1 spans six operations, where operations 2 and 6 are identical. Therefore, in Figure 4.18 (b), five bubbles are created and a number 2 is placed in the bubble representing operations 2 and 6. Figure 4.18 (b) also shows three bubbles outside the live-range boundary. These bubbles represent unique operation patterns for operations not spanned by the live-range. As shown in Figure 4.18 (c), there are three such operations that are not spanned by the current live-range. As a result, they are repre-

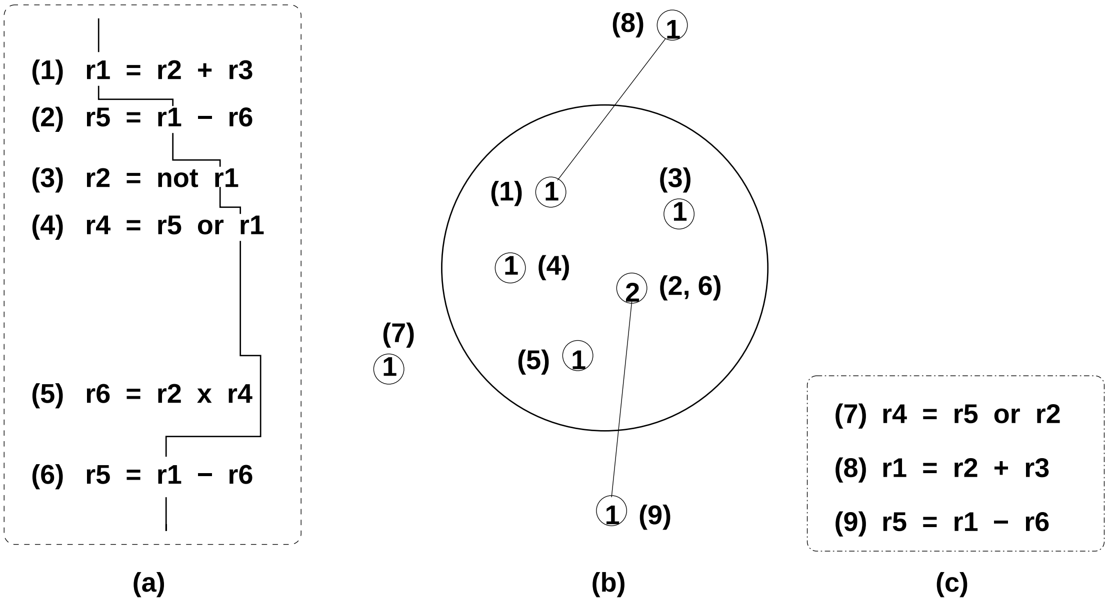

Figure 4.18:

sented as bubbles outside of the live-range boundary. An edge, connecting a bubble that is outside of the live-range boundary to a bubble inside the live-range boundary, indicates that the two bubbles represent the same unique operation pattern. For example, operations 2, 6 and 9 are identical, hence there is an edge connecting the two bubbles.

Finally, this formulation introduces a new characteristic for the live-range, called the degree of a live-range. It is defined as the total number of edges connecting bubbles inside a live-range boundary to bubbles outside of the live-range boundary. Thus, in Figure 4.18 (b), the degree of the live-range is 2.

Minimizing the total unique operation patterns can be achieved through maximizing the degree of a live-range. The example in Figure 4.18 (b) illustrates a live-range that achieves a degree of two as a result of being assigned register r1. However, if there are other available registers that can be assigned to this live-range and are capable of achieving a degree higher than two, the live-range should be recolored with the other available registers.

The effectiveness of the formulation described above is shown in Figure 4.19, where it is applied to a live-range boundary that encloses operations spanned by two live-ranges, live-range A and B. Assuming that spilling has not occurred and that both live-ranges are assigned registers, there are a total of nine bubbles inside the boundary. Four of these bubbles represent operations spanned by lr A, four represent operations spanned by lr B, and one bubble represents operations spanned by both live-ranges. Thus, the degree of this live-range boundary is four. If both live-ranges A and B are spilled, then the edges are broken because the patterns for the operations spanned by A, B and both A and B are changed when the operands are substituted with the spill registers. However, Figure 4.17, also shows that operation patterns can become the same as a result of spilling. A bubble representing an operation spanned by live-range B then has a possibility of being merged with another bubble representing an operation spanned by live-range A. However, one important observation is that spilling of live-ranges results in the lowering of the live-range degree. When the degree is lowered, edges are being broken and as a result, more patterns are being generated. To amortize the increasing number of similar operation patterns that result from spilling, a balanced amount of merging needs to take place. However, for merging to take place, enough live-ranges need to be spilled to create a bank of spill operation patterns, where subsequent operations can be made

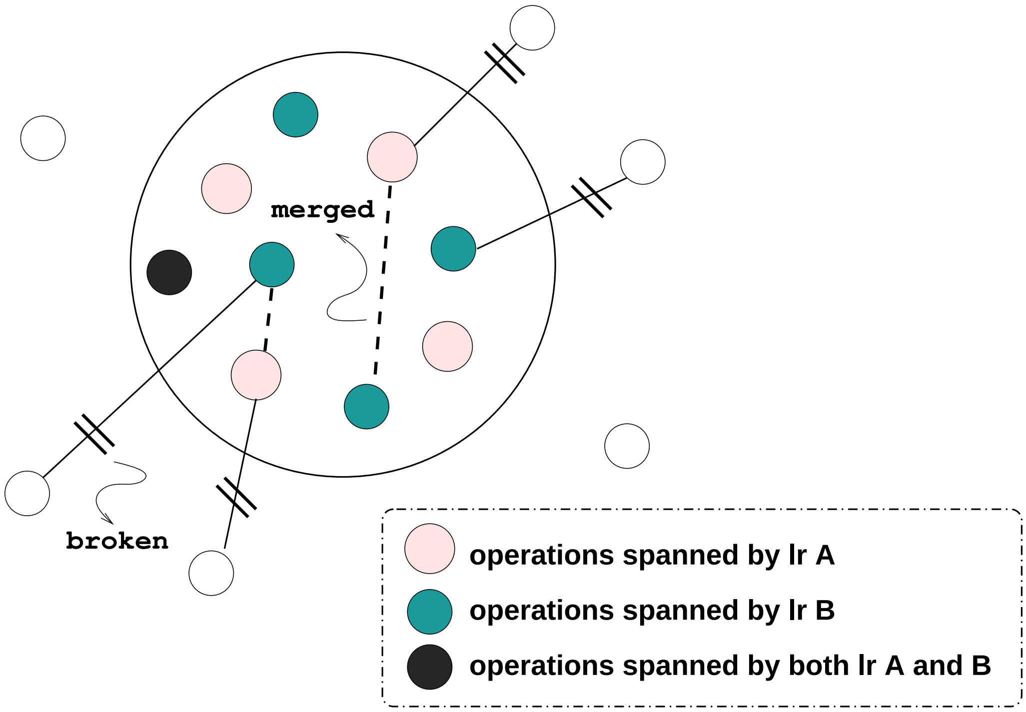

Figure 4.19:

identical to them. A simple experiment where a percentage of randomly selected live-ranges are spilled indicates that 30% of total live-ranges need to be spilled to begin to observe pattern reductions for a function where there is originally no spilling.

When minimizing the degree, the order by which a live-range is selected to undergo the recoloring process can affect subsequent live-range recoloring processes. Figure 4.20 illustrates this effect. In Figure 4.20 (a), two live-ranges, lr A and lr B are depicted. Live-range A spans four operations, three of which are shared with live-range B, indicating that live-range B also spans these three operations. Figure 4.20 (c) shows the formulation of the two

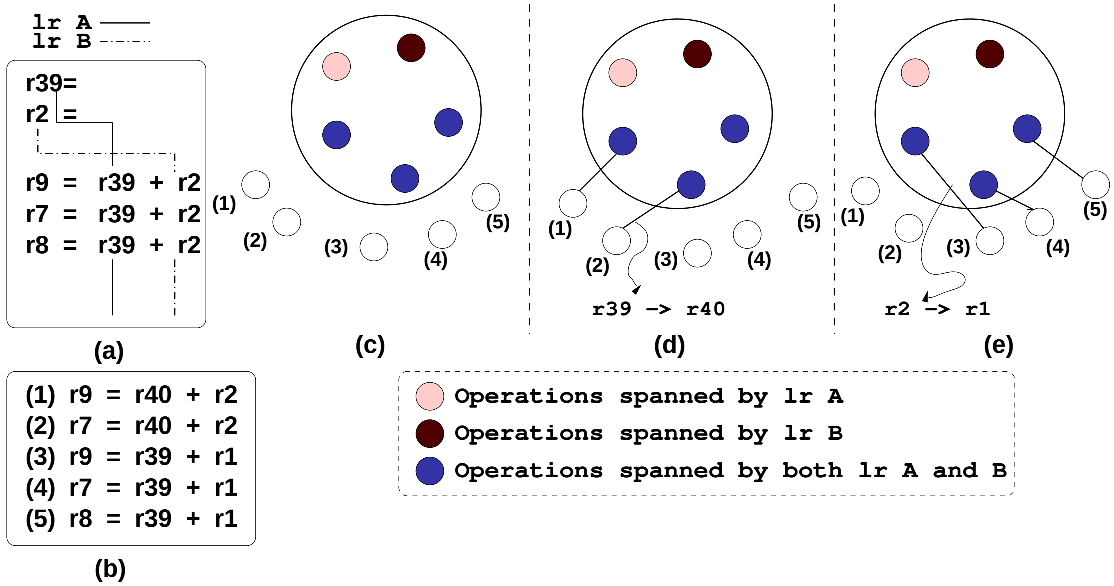

Figure 4.20:

live-ranges prior to recoloring. The degree is zero. In other words, there are ten unique operation patterns represented by ten bubbles in Figure 4.20 (c). Figure 4.20 (b) shows the existing operation patterns. If live-range A is recolored first, as shown in Figure 4.20 (d), live-range A would select register r40 instead of r39, since assigning live-range A to r40 increases the degree from zero to two. Thus, two matches are created between operations 1 and 2. On the other hand, if live-range B is recolored first, live-range B would select register r1 instead of r2, as shown in Figure 4.20 (e), since recoloring live-range B with r1 increases the degree from zero to three. It is evident then that recoloring live-range A first eliminates the opportunity for recoloring of live-range B and vice versa. Hence, the order in which live-ranges are selectively recolored plays an important role in minimizing the operation patterns.

Recoloring the Live-ranges

As briefly explained in the last section, the order in which live-ranges are recolored is critical. Two methods to recolor the live-ranges are proposed. The first method is based on a greedy approach and is referred to as the greedy recoloring technique. The second method incorporates SA into the recoloring process and is therefore referred to as recoloring with SA. Both approaches are based on the graphical formulation of live-ranges. The goal for both techniques is to minimize the total unique operation patterns by increasing the degree of each live-range through recoloring.

The main loop of the greedy recoloring technique is shown in Figure 4.21 and the procedures called by the main loop are shown in Figure 4.22. The greedy algorithm assigns a live-range with another available register that is capable of increasing the degree of the live-range. The main loop iterates over each live-range as shown in step 1. If the live-range is spilled by the original allocator (step 2), it remains as spilled and is not considered for recoloring. In step 3, the unused caller saved and callee saved registers for the live-range are gathered and represented by bit vectors *available_caller_regs* and *available_callee_regs*. Steps 4, 5, and 6 determines whether it is more beneficial to recolor the live-range with a caller-saved or a callee-saved register. Once the pool of registers is determined, the algorithm proceeds to recolor the live-range. The RECOLOR procedure is provided in Figure 4.22. As shown in steps 9, 10 and 11, the purpose of the RECOLOR procedure is to select a register from the set of available registers that is capable of increasing the degree of the live-range. The degree of the live-range achieved by each available register can be easily determined by steps 1 to 8 of the

RECOLOR procedure.

Figure 4.23 shows the algorithm for the recoloring with SA technique. Figure 4.24 shows the procedures called by the main annealing routine. The initial state is the coloring produced by the baseline allocator. The total unique operation patterns is used as the cost. Step 3 of the algorithm builds the pattern database which stores the unique operation patterns in the allocated code. The equations for computing the initial temperature as shown in Procedure INITIAL_TEMPERATURE of Figure 4.24 are derived from experiments which are annealing runs using functions of different sizes. The EXIT_CRITERION is fulfilled when the costs stabilize over three outer while iterations. Each new state is derived by randomly recoloring 25% of the total number of live-ranges. The RECOLOR function used in step 5 is shown in Figure 4.22 where a live-range is recolored to achieve a higher degree. The equation which updates the temperature is also derived from experiments.

Results and Discussion

Figure 4.25 presents the total unique operation patterns of ten functions selected from benchmarks compress, li, go, m88ksim and vortex. In each case, three results are presented showing the unique operation patterns created by the three coloring techniques: coloring with a database, greedy recoloring and recoloring with SA. The results are normalized to the patterns created using the coloring with a database technique. Seven of the ten functions are able to show on average, a 2% pattern reduction. However, three functions, *Draw7*, *iscaptured*, and *lo* show increases in patterns. These three functions belong to benchmarks, *vortex*, *go* and *m88ksim* respectively. Figure 4.26 shows the

```
Input: (1) life−ranges that have been assigned colors by the baseline allocator
       (2) a pattern database storing the unique operation patterns

Output: life−ranges with new color assignments

Definition:
     (1)  The BENEFIT function compares the benefits of assigning either a
          caller−saved or a callee−saved register to a live−range.  The pool
          with higher benefit is returned.
     (2)  The functions of UNUSED_CALLER_COLORS and
          UNUSED_CALLEE_COLORS determine the available caller or
          callee−saved registers for a live−range by examining the colors
          of its interfering live−ranges, or in another word, the colors of
          its neighbouring nodes in the interference graph.

1.  loop over each input live−range
         current_lr <= life−range currently being considered

      2.  if (current_lr is marked as spilled)  then
                   continue to examine the next life−range
          end if
      3.  available_caller_regs <= UNUSED_CALLER_COLORS(current_lr)
          available_callee_regs <= UNUSED_CALLEE_COLORS(current_lr)
      4.  if (BENEFIT(current_lr)  == CALLER_SAVED)  then
                   if (available_caller_regs.ones_count() > 0) then
                             RECOLOR(current_lr, available_caller_regs)
                 else if (available_callee_regs.ones_count() > 0  and
                         is_callee_saved_reg(original_color) == TRUE)
                            RECOLOR(current_lr, available_callee_regs)
                 end if
      5.  else if (BENEFIT(current_lr) == CALLEE_SAVED)
                  if (available_callee_regs.ones_count() > 0)  then
                            RECOLOR(current_lr, available_callee_regs)
                 else if (available_caller_regs.ones_count() > 0 and
                         is_caller_saved_reg(original_color) == TRUE)
                            RECOLOR(current_lr, available_caller_regs)
                 end if
      6.  else
                  RECOLOR(current_lr, available_caller_regs + available_callee_regs)
          end if

          // the new color is different from the original
          if (current_lr has been recolored)  then
                  update pattern database
          end
   end loop
```

Figure 4.21: Algorithm: greedy recoloring technique.

```
Procedure UNUSED_CALLER_COLORS(live−range lr) returns a bit vector
        available_caller_saved_regs <= all caller−saved registers
        loop over neighbours of the life−range lr
                if (is_caller_saved_reg(neighbour’s color)) then
                        remove neighbour’s color from available_caller_saved_regs
                end if
        end loop
        return available_caller_saved_regs
end Procedure

Procedure RECOLOR(live−range lr, bit_vector available_colors)
        UNCOLOR live−range lr
        operation_spanned <= 0
    1.  loop over operations spanned by lr
    2.          current_op <= operation currently being considered
    3.          current_environment <= ENVIRONMENT(current_op)
    4.          used_colors_array[operation_spanned] <=
                                        LOOK_UP(current_environment)
    5.          operation_spanned++
        end loop

        loop over each available color
                current_color <= available color currently being considered
                degree <= 0
        6.      for i:= 0 to operation_spanned − 1
        7.              if (used_colors_array[i].is_one(current_color) == TRUE)
        8.                      degree++
                        end if
                end for
        9.      if (degree > highest_degree)  then
        10.             highest_degree <= degree
        11.             color_with_highest_degree <= current_color
                end if
        end loop
        RECOLOR lr with color_with_highest_degree
end Procedure
```

Figure 4.22: Functions called by the main loop of the greedy recoloring algorithm.

```
1.  state  <=  color assignments generated by the baseline allocator
2.  previous_cost <= total number of unique operation patterns created by
                          the initial coloring
3.  build pattern database
4.  temperature <= INITIAL_TEMPERATURE(previous_cost)
while (EXIT_CRITERION(previous_cost) == FALSE)
        current_iteration <= 0
        while (INNER_LOOP_CRITERION(current_iteration) == FALSE)
                current_iteration++
5.              new_state <= randomly select 25% of the total number of
                                  life−ranges to RECOLOR
              cost <= total number of unique operation patterns for
                          the new state
              if (ACCEPT(cost, prev_cost, temperature)) then
                          state <= new_state
              end if
        end while
        temperature = UPDATE_TEMPERATURE(temperature)
end while
```

Figure 4.23: Algorithm: recoloring with simulated annealing.

storage requirements for the five benchmarks using the proposed recoloring techniques. Baseline packer is used to obtain the decoder memory sizes. In each case, the decoder memory size is normalized to the size obtained when coloring is performed by the original allocator. As shown in Figure 4.26, increases in decoder memory sizes for benchmarks m88ksim, go and vortex are observed. In benchmark vortex, the increase is approximately 10%. On the other hand, an approximately 3% reduction is observed in benchmark compress.

Though the two recoloring techniques are capable of achieving a small percentage of pattern reduction, there is no significant reduction in terms of decoder memory sizes. As discussed in Section 4.2, minimizing operation patterns can lead to a decrease in decoder memory size. However, if the

```
Procedure UPDATE_TEMPERATURE(float T) returns float
        return (0.99 x T)
end Procedure
Procedure INITIAL_TEMPERATURE(float cost) returns float
        max_range = cost/60
        return (−max_range/log(0.2))
end Procedure
Procedure INNER_LOOP_CRITERION(int iteration) returns boolean
        if (iteration < 100) then
                return false
        else
                return TRUE
        end if
end Procedure
Procedure EXIT_CRITERION(float cost) returns boolean
        if (cost is the same as the cost parameters from the previous two calls)
                return TRUE
        else
                return FALSE
        end if
end Procedure
Procedure ACCEPT(float cost, float previous_cost, float temperature) returns boolean
        delta_c <= cost − previous_cost
        y <= exp(−delta_c/temperature)
        random_number = a random floating point number in the range of [0, 1)
        if (random_number < y)
                return TRUE
        else
                return FALSE
        end if
end Procedure
```

Figure 4.24: Functions called by the main annealing routine.

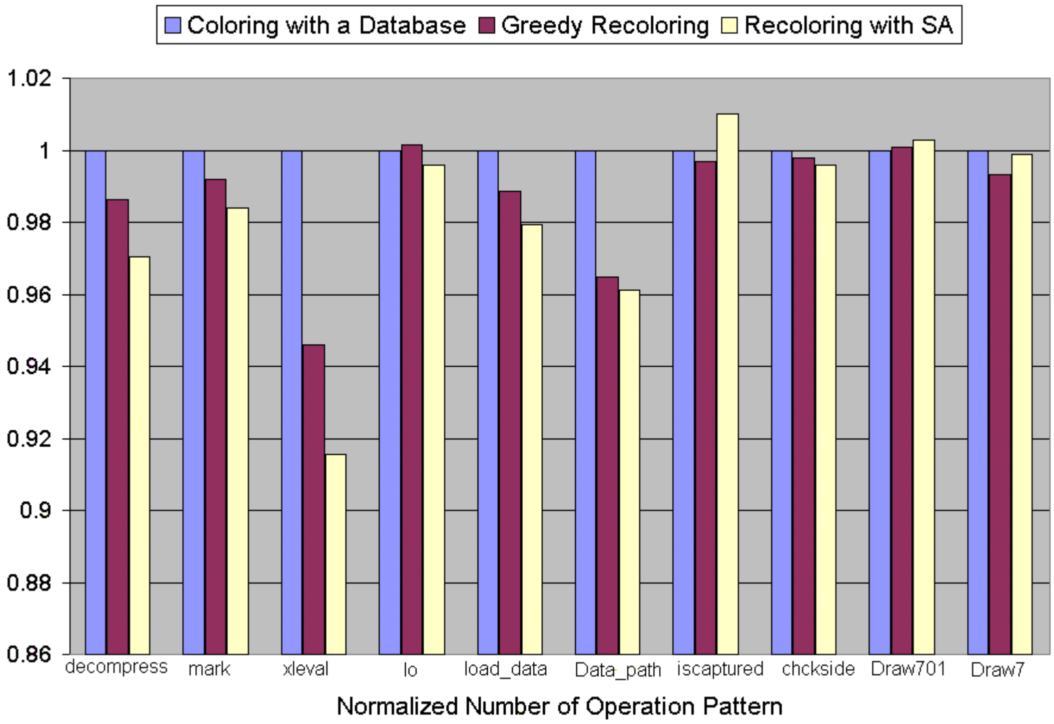

Figure 4.25: Unique operation pattern reduction.

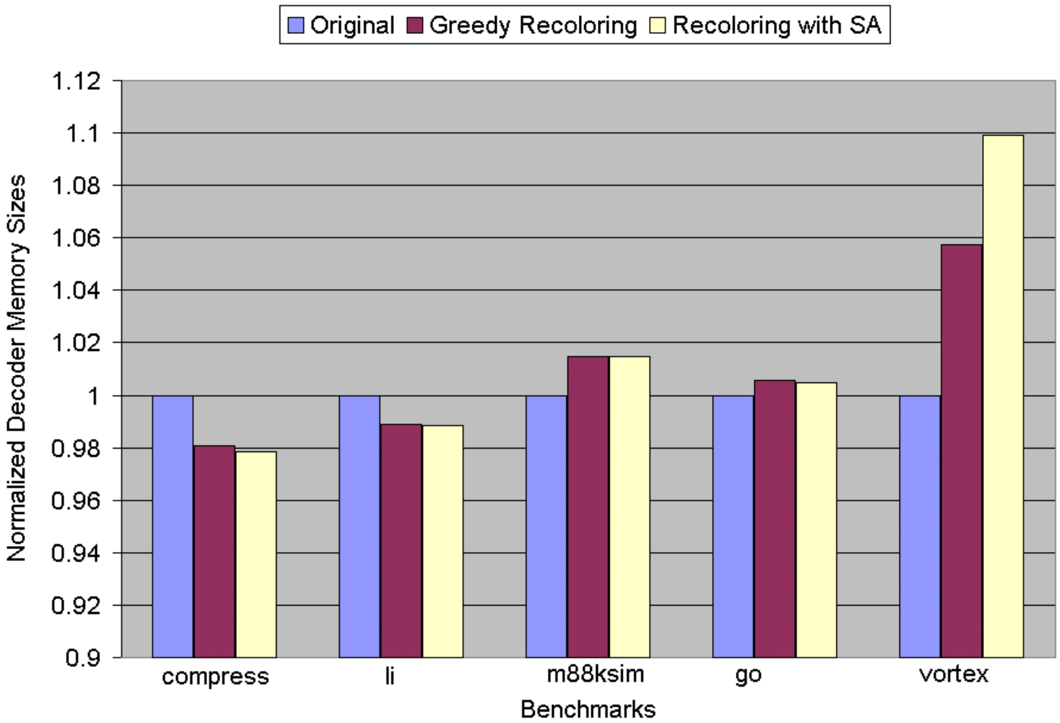
Original
Greedy Recoloring
Recoloring with SA
Normalized Decoder Memory Sizes
1.12
1.1
1.08
1.06
1.04
1.02
1
0.98
0.96
0.94
0.92
0.9
compress
li
m88ksim
go
vortex
Benchmarks

Figure 4.26:

pattern reduction is as small as 2reduction in decoder memory size can be insignificant. The reason is because the small percentage of pattern reduction can be easily lost when operations undergo instruction scheduling. When an operation undergoes instruction scheduling, it may be scheduled along with other operations to form a long instruction. Two identical operations can no longer be packed into the same memory location if they are scheduled with different operations. Also, the cost driving the annealer does not take scheduling into account. As shown in Figure 4.23 (a), the number of unique operation patterns is used as the cost for a coloring state.

Lastly, the recoloring techniques focus on minimizing the unique type I and II operation patterns, excluding the spill operation patterns. However, type I operations only account for a small fraction of total operations, and hence minimizing type I operation patterns cannot lead to significantly reduced pattern numbers. Furthermore, type II operation patterns are difficult to minimize as they have operands that are unique constants as opposed to register operands.

As future work, characteristics of each benchmark can be further analyzed, focusing on the impact of pattern reduction on instruction packing. Also, the cost of the annealer can explore ways to take instruction scheduling into account.

Summary

An optimized register allocation is a highly constrained graph coloring process where each live-range is selected according to a predefined priority and colored with the most beneficial register. Hence, a graphical formulation is introduced and under this framework, SA is incorporated into the coloring process. In particular, SA is required to find an order in which live-ranges are recolored that can lead to a reduction in the total unique operation patterns. Under this framework, a greedy recoloring algorithm is also proposed.

The experimental results show that both recoloring techniques are only capable of achieving a small reduction in unique operation patterns, and are unable to achieve any reduction in the final decoder memory size.

4.4 Conclusion

This chapter examines three different methods to achieve a denser decoder memory. The first technique improves existing instruction packing algorithms by introducing binding-shifting and commutative transform into the algorithm. Applying both techniques in the algorithm result in the utilization of 100% of the decoder memory and the achievement of up to 3.2% in size reduction. The second method restructures the stack, which in turn minimizes the total number of unique constant operands. This improvement reduces the number of unique operation patterns created and reduces up to 16.8% of the decoder memory. The last approach minimizes the unique type I and type II operation patterns through a modified register allocation process. Although the idea is novel, unfortunately, the two proposed recoloring approaches do not achieve any significant decoder memory reduction. Thus, it is evident that it is the use of binding-shifting and commutative transform in instruction packing algorithms along with stack restructuring that ultimately contributes to the obtainment of a denser decoder memory.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This thesis proposes various techniques to achieve a denser and smaller decoder memory used for storing VLIW instructions in the two-level memory framework. The techniques examine three orthogonal directions to achieve code compaction.

The first direction focuses on improving the existing instruction packing algorithm. Two techniques, binding-shifting and commutative transform, are proposed and incorporated into the existing algorithm. The techniques are evaluated using the assembly code generated by the Trimaran compiler for a set of SpecInt95 benchmarks. The results show that the techniques are effective in achieving on average 2.4% size reduction, and 100% memory utilization for the set of benchmarks without affecting the performance.

The second direction focuses on restructuring the register save area of the stack. Furthermore, two stack related optimization techniques are proposed and evaluated. The first technique stops the allocator from assigning a memory location to every local variable in the function. Instead, only variables that cannot be allocated to registers are assigned home locations on the stack. Hence, the locations are assigned during the coloring process to variables whose live-ranges are spilled. The second technique further minimizes the stack size by assigning the same stack location to variables whose live-ranges do not interfere. The techniques are capable of achieving a maximum of 16.8% reduction in decoder memory size. This is a clear indication that minimizing the number of unique constant operands is a step towards achieving a smaller decoder memory. There is also no effect on the execution performance.

The third direction focuses on using register allocation to minimize the number of unique operations. Three techniques are proposed. The first technique uses a pattern-aware color selection process where a database is used to store the existing operation patterns. Frequency-based color selection is proposed as an example of a pattern-aware color selection process. The second technique is a greedy live-range recoloring technique used to minimize the number of unique operations that have only register operands (type I) and operations that have both register and constant operands (type II). The third technique incorporates simulated annealing into the live-range recoloring process. Both techniques are based on the graphical representation of a live-range and the operations spanned by the live-range. Unfortunately, none of the three techniques are able to achieve a reduction in terms of total decoder memory size. The main reasons are summarized as follows.

The three techniques focus on manipulating the register operands of type I and II operations through coloring. Through assigning the colors in the right way, operation patterns can be minimized. However, the type of operation that can truly benefit from these techniques is the type I operation because the allocator has full control of all the operands of a type I operation. However, type I operation patterns only account for a small fraction of the total operation patterns. For example, in the first cluster of benchmark vortex, the number of type I operation patterns is only 4% of the total operation patterns and less than 1% in the second cluster, as shown in Table 3.1. As for the type II operations, patterns cannot be minimized solely by manipulating the register operands because type II operations also have operands that are constants. The constant operands are the main cause of the large number of unique type II operation patterns, not the register operands. Due to these reasons, the three techniques using register allocation are not effective in achieving a smaller decoder memory.

In conclusion, it is evident that it is the use of binding-shifting and commutative transform in instruction packing algorithms along with stack restructuring that ultimately contributes to the obtainment of a denser decoder memory.

5.2 Future Work

For future work, ways to further minimize the number of unique constant operands can be explored, focusing on minimizing the branch related constants, e.g. branch labels and offsets. After the constants are minimized, other words, when the majority of the operands are registers, then the register recoloring techniques proposed by this thesis can become useful. The process of recoloring with annealing can then be fine-tuned. Fine-tuning the annealing process can involve a redefinition of the cost so that instruction scheduling is taken into account, and modifications of the rules which model an optimized register allocation process. The initial temperature and the lowering of the temperature can also be made adaptable to functions with different sizes, leading to faster convergence.

Another area for future exploration is in instruction packing when a limited size of decoder memory is given. Methods to selectively pack instructions and the effect on performance can be studied.

Overall, better understanding of the characteristics of each benchmark is needed and also, the techniques need to be evaluated using other types of benchmarks.

Lastly, ways to apply the stack optimization techniques to other compiler systems should be explored. More efforts should be directed to acquiring a broad understanding of other existing stack optimization techniques.

Appendix A

A.1 Mapping to Clique Partitioning

The following paragraph provides a simple argument for the validity of mapping the instruction packing problem into a clique partitioning problem.

A simple argument can be presented as follows. Assume nodes A, B, C form a clique, select any pair from the three nodes, e.g., nodes X and Y. Pack nodes X and Y into a word and there are only two possibilities for the fields of the packed word: a NOOP, or an operation. If the field of the packed word is an operation, then it must have been an operation belonging to either X, or Y or to both. Upon packing the third node, referred to as Z, with the packed word, the NOOP fields in the packed word can pack unquestionably with corresponding fields of Z. Upon encountering fields containing operations in both the packed word and in Z, the operations in the packed word and in Z must be identical as the initial assumption states that Z must be a subset of both X and Y.

A.2 Sizes of Each Individual Decoder Memory Bank

Tables A.1 to A.5 show the sizes of each decoder memory banks achieved by the various packing techniques described in Section 4.1. The unit for the sizes is in *word* which equals four bytes.

Benchmarks	ialu0	ialu1	falu0	falu1	memu0	memu1	bru0	total	empty	% utilization
compress	1760	1601	43	6	583	425	188	4604	82	98.22
li	4984	4712	38	2	1007	732	341	11816	192	98.34
m88ksim	13217	12585	42	26	2621	1962	711	31164	332	98.93
go	14825	14054	0	0	1965	1448	359	32651	168	99.49
perl	25821	24173	408	4	4728	3759	1363	60256	696	98.84
vortex	25598	24514	11	0	4552	3564	374	58613	122	99.79

Table A.1: Sizes of Each Decoder Memory Bank Required using Subset Packing

Benchmarks	ialu0	ialu1	falu0	falu1	memu0	memu1	bru0	total	empty	% utilization
compress	1667	1666	42	6	465	465	162	4473	0	100
li	4817	4816	34	2	740	739	286	11434	0	100
m88ksim	12831	12831	42	26	2040	2040	587	30397	0	100
go	14349	14348	0	0	1533	1533	286	32049	0	100
perl	24796	24796	302	4	3783	3783	1156	58620	0	100
vortex	24923	24923	8	0	3898	3897	318	57967	0	100

Table A.2: Sizes of Each Decoder Memory Bank Required using Augmented Subset Packing

Benchmarks	ialu0	ialu1	falu0	falu1	memu0	memu1	bru0	total	empty	% utilization
compress	1666	1666	42	6	465	465	162	4472	0	100
li	4816	4816	34	2	740	739	286	11433	0	100
m88ksim	12817	12816	42	26	2040	2040	587	30366	0	100
go	14332	14331	0	0	1533	1533	286	32015	0	100
perl	24777	24777	302	4	3783	3783	1156	58582	0	100
vortex	24903	24902	8	0	3898	3897	318	57926	0	100

Table A.3: Sizes of Each Decoder Memory Bank Required using Augmented Commutative Subset Packing

Benchmarks	ialu0	ialu1	falu0	falu1	memu0	memu1	bru0	total	empty	% utilization
compress	1760	1758	1696	487	583	579	574	7437	2872	61.38
li	4984	4984	3987	15	1027	1027	1026	17050	5290	68.97
m88ksim	13217	13217	12333	2522	2665	2664	2665	49283	18081	63.31
go	14825	14825	0	0	1983	1983	1983	35599	2915	91.81
perl	25821	25820	24789	2781	4824	4824	4822	93681	33311	64.44
vortex	25598	25598	24786	0	4604	4603	4555	89744	30903	65.57

Table A.4: Sizes of Each Decoder Memory Bank Required using Packing by Region

Benchmarks	ialu0	ialu1	falu0	falu1	memu0	memu1	bru0	total	empty	% utilization
compress	1760	1760	1687	505	586	583	577	7458	2933	60.67
li	4984	4984	4040	2	1014	1014	1013	17051	5437	68.11
m88ksim	13217	13217	12382	2510	2623	2622	2623	49194	18364	62.67

Table A.5: Sizes of Each Decoder Memory Bank Required using Clique Partitioning

Bibliography

[1] Mazen A. R. Saghir. *Application-Specific Instruction-Set Architectures for Embedded DSP Applications.* PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, 1998.

[2] Safwat G. Zaky V. Carl Hamacher, Zvonko G. Vranesic. *Computer Organization.* The McGraw-Hill Companies, Inc., fourth edition, 1996.

[3] Robert Muth Saumya K. Debray, William Evans. Compiler Techniques for Code Compression. Technical Report TR99-07, Department of Computer Science, Unversity of Arizona, 1999.

[4] Nathaniel McIntosh K.D. Cooper. Enhanced Code Compression for Embedded RISC Processors. In *Proceeding of the '99 International Conference on Programming Language Design and Implementation*, 1999.

[5] Brenda S. Baker. A Program for Identifying Duplicated Code. In *Proceedings of the 24th Symposium on the Interface*, pages 49–57, March 1992.

[6] Alan L. Wendt Christopher W. Fraser, Eugene W. Myers. Analyzing and Compressing Assembly Code. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, volume 19, pages 117–121, June 1984. SIGPLAN Notices.

[7] Kurt Keutzer Stan Y. Liao, Srinivas Devadas. Code Density Optimization for Embedded DSP Processors Using Data Compression Techniques. In *Proceeding of Chapel Hill Conference Advanced Research in VLSI*, March 1995.

[8] Billions of Operations Per Second Inc. Methods and Apparatus For Scalable Instruction Set Architecture with Dynamic Compact Instructions, 2000. United States Patent.

[9] John L. Hennessy Fred C. Chow. The Priority Based Coloring Approach to Register Allocation. *ACM Transaction on Programming Languages and Systems*, 12(4):501–536, October 1990.

[10] Yanti Sugono Sami Khuri, Tim Walters. A Grouping Genetic Algorithm for Coloring the Edges of Graphs. In *Proceedings of the 2000 ACM Symposium on Applied Computing*, volume 1, pages 422–427, 2000.

[11] Gary W. Grewal. *Enhanced Genetic Algorithms and their Application in Retargetable Code Generation*. PhD thesis, Department of Electrical and Computer Engineering, University of Guelph, 1998.

[12] Wen-mei W. Hwu, Richard E. Hank, David M. Gallagher, Scott A. Mahlke, Daniel M. Lavery, Grant E. Haab, John C. Gyllenhaal, David I. August. Compiler Technology for Future Microprocessors. In *Proceedings of the IEEE*, volume 83, pages 1625–1640, December 2000.

[13] B. Ramakrishna Rau Vinod Kathail, Michael S. Schlansker. HPL-PD Architecture Specification: Version 1.1. Technical Report HPL-93-80, Hewlett Packard, 1994.

[14] D.I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, July 1998.

[15] Hansoo Kim. *Region-based Register Allocation for EPIC Architectures*. PhD thesis, Department of Computer Science, New York University, 2000.

[16] Standard Performance Evaluation Corporation. Spec cpu95 benchmarks, 1995. http://open.specbench.org/osg/cpu95/.

[17] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register Allocation via Coloring. *Computer Languages*, 6:47 – 57, 1981.

[18] David A. Patterson John L. Hennessy. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1996.

[19] Sean Hsien en Peng. UTDSP: A VLIW Programmable DSP Processor. Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, 1999.

[20] The mapping of the instruction packing problem into a clique partitioning problem was originally suggested by Tor Aamodt at the University of Toronto.

[21] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*, pages 66–67. The McGraw-Hill Companies, Inc., 1994.

[22] Johnny Burlin. Optimizing Stack Frame Layout for Embedded Systems. Master's thesis, Department of Computer Science, Uppsala University Sweden, 2000.

[23] Robert P. Wilson, el al. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers, 1994. http://suif.stanford.edu/suif/suif1/suif-overview/suif.html.

[24] Alexander Marquardt Vaughn Betz, Jonathan Rose. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.