THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

USING DE-OPTIMIZATION TO RE-OPTIMIZE CODE

By

STEPHEN R. HINES

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Summer Semester, 2004

The members of the Committee approve the thesis of Stephen R. Hines defended on April 30, 2004.

David Whalley
Professor Directing Thesis

Gary Tyson
Committee Member

Xin Yuan
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

To Mom, Dad, and Frank . . .

ACKNOWLEDGMENTS

I am very grateful for the help of my advisor, Dr. David Whalley. Without you, this thesis would not have been possible. Thank you for believing in me, as well as inspiring me to work hard to achieve my goals.

I would also like to thank the other members of the Compilers Group (Prasad Kulkarni, Bill Kreahling, Clint Whaley, Wankang Zhao) for their assistance. This work would have been extraordinarily difficult without your insight and your friendship.

I would like to extend a big thanks to my family and friends for their unwavering love and support. You may not understand all of the complexities involved in my research, but I certainly learned that you are always willing to listen to me. I am truly blessed to have each of you in my life.

TABLE OF CONTENTS

List of Tables vii

List of Figures viii

Abstract ix

1. INTRODUCTION 1

2. EXPERIMENTAL SETTING 5

2.1 VPO Interactive System for Tuning Applications (VISTA) 5
2.1.1 Very Portable Optimizer (VPO) 6
2.1.2 Searching for Effective Phase Sequences 6
2.2 Hardware and Software Platform 7
2.2.1 MiBench Embedded Applications Benchmark Suite 9
2.2.2 Generating Assembly Code 10
2.3 Experimental Test Plan 11

3. DEVELOPMENT OF ASM2RTL TOOLS 14

3.1 Assembly Translation and Re-optimization 14
3.2 Preservation of Program Semantics 16
3.2.1 Information Loss 16
3.2.2 Local Variables 17
3.2.3 Calling Conventions 18
3.3 Implementation Strategy 19
3.4 Translation Difficulties 21
3.4.1 Memory Consistency 21
3.4.2 Following Calling Conventions 23
3.4.3 Translation Tradeoffs 24
3.5 Additional Translators 25

4. DEVELOPMENT OF DE-OPTIMIZATIONS 27

4.1 Loop-Invariant Code Motion 27
4.1.1 Motivation for Undoing Loop-Invariant Code Motion 28
4.1.2 De-optimizing Loop-Invariant Code Motion 29
4.2 Register Allocation and Register Assignment 32
4.2.1 Motivation for Undoing Register Allocation 33
4.2.2 Register Interference Graphs 34

4.2.3 De-optimizing Register Allocation 36
4.2.4 Example De-optimization of Register Allocation 39
4.3 De-optimization Difficulties 43
4.3.1 Calling Conventions Revisited 44
4.3.2 Code Expansion and Global Variable Offsets 45

5. REVERSE COPY PROPAGATION 47

5.1 Motivation for Reverse Copy Propagation 47
5.2 Implementing Reverse Copy Propagation 50
5.3 Reverse Copy Propagation Example 53

6. EXPERIMENTAL TESTING 56

6.1 Experimental Results 56
6.2 Discussion of Results 58
6.2.1 Re-optimization Benefits 59
6.2.2 Effectiveness of Register Re-assignment 59
6.2.3 Genetic Algorithm Behavior 59
6.2.4 Dynamic Execution Statistics 60

7. RELATED WORK 62

8. ADDITIONAL BENEFITS OF THIS WORK 65

8.1 Use of ASM2RTL in DSP-driven Labs 65
8.2 Detecting Homomorphic Code in VISTA 66

9. FUTURE WORK 68

9.1 More Extensive Experiments 68
9.2 Improving De-optimization 69

10. CONCLUSIONS 71

REFERENCES 73

BIOGRAPHICAL SKETCH 75

LIST OF TABLES

2.1 MiBench Benchmarks Used for Experiments 9
2.2 GCC Optimization Flags 11
2.3 VISTA Genetic Algorithm Candidate and Required Phases 13
6.1 Effect of De-optimization on Static and Dynamic Instruction Count 57

LIST OF FIGURES

1.1 Assembly De-optimization and Re-optimization . . . 3
2.1 VISTA Genetic Algorithm Search Parameters . . . 7
2.2 VISTA Genetic Algorithm Search (In Progress) . . . 8
3.1 Iterative Re-optimization of Assembly Code . . . 16
3.2 Local Variable Confusion . . . 18
3.3 ASM2RTL Program Structure . . . 20
3.4 Local Variable Reconstruction . . . 22
4.1 De-optimize Loop-Invariant Code Motion . . . 30
4.2 De-optimizing Loop-Invariant Code Motion . . . 31
4.3 Constructing a Register Interference Graph . . . 35
4.4 De-optimize Register Allocation . . . 38
4.5 Dequeue from Dijkstra Benchmark . . . 39
4.6 Dequeue Prior To De-optimizing Register Allocation . . . 40
4.7 Dequeue After De-optimization of Register Allocation . . . 41
4.8 Dequeue After Re-performing Register Assignment . . . 43
4.9 Dequeue After Additional Optimizations . . . 44
5.1 Register Re-assignment with Keymatch . . . 49
5.2 Reverse Copy Propagation . . . 51
5.3 Backward Scan . . . 52
5.4 Forward Scan . . . 53
5.5 Reverse Copy Propagation with Keymatch . . . 55
8.1 Two Functions with Homomorphic Code . . . 67

ABSTRACT

The nature of embedded systems development places a great deal of importance on meeting strict requirements in areas such as static code size, power consumption, and execution time. Due to this, embedded developers frequently generate and tune assembly code for applications by hand. The phase ordering problem is a well-known problem affecting the design of optimizing compilers. VISTA is an optimizing compiler framework that employs iteration of optimization phase sequences and a genetic algorithm search for effective phase sequences in an effort to minimize the effects of the phase ordering problem. Hand-generated code is susceptible to an analogous problem to phase ordering, but there has been little research in mitigating its effect on the quality of the generated code. One approach for adjusting the phase ordering of such previously optimized code is to de-optimize the code by undoing the potential work done by prior optimization phases. This thesis presents an extension of the VISTA framework for investigating the effect and potential benefit of performing de-optimization before re-optimizing assembly code. The construction of a translator tool suite for the purpose of converting assembly code to the VISTA RTL input format is discussed. The design and implementation of algorithms for de-optimization of both loop-invariant code motion and register allocation, along with results of performing experiments regarding de-optimization and re-optimization of previously generated assembly code are also presented.

CHAPTER 1

INTRODUCTION

The phase ordering problem is a long-standing problem involved in the development of compilers and related tools [20]. Simply put, the phase ordering problem is that there exists no single sequence of optimization phases that will produce optimal code for every function in every application on every architecture. Different optimizations can enable or disable further optimizations depending on the characteristics of the current function being compiled as well as the target architecture [21]. These enabling or disabling factors can greatly impact the design and implementation of optimizing compilers. One of the most critical enabling or disabling factors is register pressure. Many optimization phases consume registers, thus increasing register pressure. Optimization phases performed after this point will then have fewer available registers to use. Depending on the chosen optimization phase order, later phases may be prohibited from having any effect at all. Embedded systems are even at greater risk of trouble due to the phase ordering problem, since these systems typically have non-orthogonal instruction sets and fewer registers, contributing to even greater register pressure.

One approach towards minimizing the effects of the phase ordering problem is to produce a compiler with the ability to apply phases repeatedly and in any given order. The Very Portable Optimizer (VPO) was developed in an attempt to provide these exact features [1]. VISTA (VPO Interactive System for Tuning Applications) is an enhancement developed for VPO to provide a graphical interactive compilation framework which application developers can use to finely tune generated code [22]. VISTA provides static and dynamic measurement information to allow the programmer to more effectively guide the optimization process [13]. Additionally, the VISTA framework supports automatic tuning of code through the use of static and dynamic profile data combined with a genetic algorithm search for effective phase

sequence orderings [15]. All of these features make VISTA an attractive environment for the study of optimizing compiler technology.

Embedded devices are experiencing a great surge in popularity. As the demand for embedded devices increases, so too does the demand for embedded software development. With embedded software development, size and timing constraints are both more important and more stringent than they are in traditional software. This increased focus forces a great deal of embedded applications development to be done using assembly language, since the ability to hand-tune code typically produces smaller and faster executables than comparable high-level languages with good optimizing compilers.

Hand-generated assembly code may appear to be an adequate solution for the requirements of embedded software development, however it is still subject to the phase ordering problem albeit in a slightly different manner. Clearly the majority of optimization phases need not be performed in exactly the same manner by an assembly programmer who is hand-coding a function. Transformations from different optimization phases can be mixed and combined at many different points during the code generation and hand tuning process. Thus there is not truly an explicit phase ordering being applied to the function. Instead, the programmer modifies the code based on intuitions and educated decisions in an attempt to decrease the static and dynamic instruction counts. However these judgments are clearly similar to phase ordering decisions made by an optimizing compiler, so it is possible that a better solution exists even with hand-tuned assembly code.

It is thus natural to attempt to alleviate the phase ordering problem with hand-generated assembly code by making some modifications to the facilities available in the VISTA framework. First a translator must be constructed to convert the assembly code to an input format which VISTA can understand. To handle the phase ordering problem, the concept of de-optimization will be applied. Prior optimizations that affect the phase ordering problem can be undone in a safe manner, so that different phase sequences can then be applied and tested. For the purpose of this thesis, both *loop-invariant code motion* and *register allocation* will be de-optimized since they have a great impact on register pressure and thus the phase ordering problem. Other optimizations were not selected because they either do not have much of an impact on the phase ordering problem (*branch chaining*), or there is no way to reconstruct the necessary information (*dead assignment elimination*). After performing

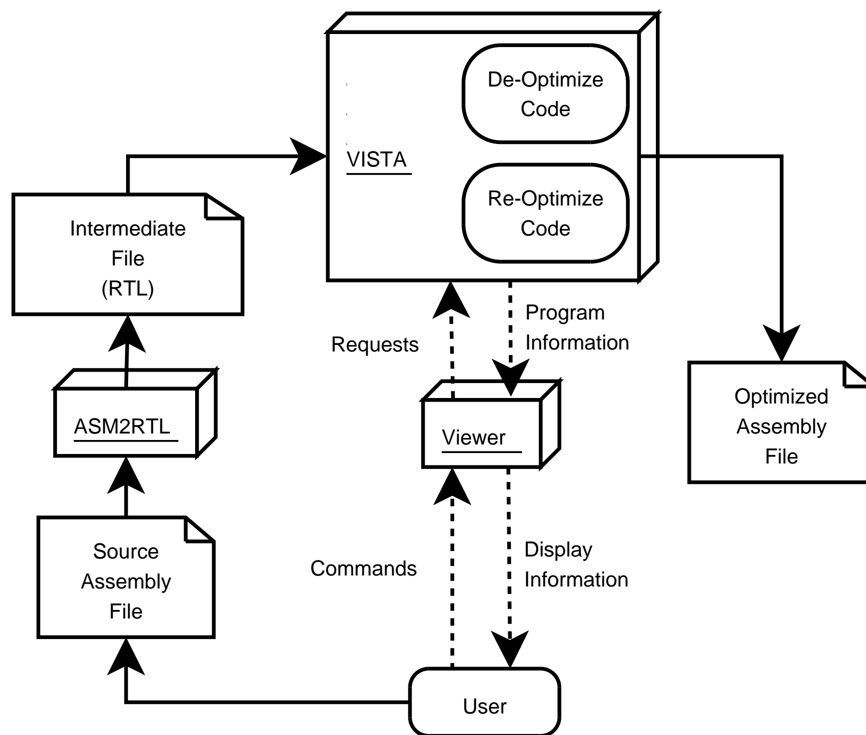


Figure 1.1: Assembly De-optimization and Re-optimization

the required de-optimizations, VISTA can then be instructed to automatically retune the de-optimized code sequence using its genetic algorithm search for effective optimization sequences. Figure 1.1 shows the process of re-optimizing assembly code with VISTA.

The rest of this thesis can be broken down as follows: Chapter 2 presents additional preliminary information, including a detailed look at the VISTA framework, which will be used for all testing. The next chapter discusses the process of assembly translation, including the implementation of the ASM2RTL translator suite for converting assembly source code to the VISTA RTL format. Chapter 4 focuses on the actual design and development of de-optimizations within the VISTA framework. Chapter 5 describes the optimization *reverse copy propagation*, which was added to VISTA to improve de-optimized and re-optimized code. The results of performing the proposed experiments are then examined, along with a discussion of the analyzed data. Related work is then reviewed for the areas of assembly translation and de-optimization. Additional benefits of the work done in this thesis are highlighted in Chapter 8. Chapter 9 considers some of the potential future improvements

that can be made to the de-optimization strategy. The final chapter contains concluding remarks regarding the use of de-optimizations for re-optimizing code.

CHAPTER 2

EXPERIMENTAL SETTING

The general setup for the de-optimization experiment is described in detail in this chapter. The first section discusses VISTA, the compiler framework for which the proposed de-optimizations will be implemented and tested. The next section contains information concerning the necessary hardware and software for adequately testing the de-optimizations. This also includes detailed descriptions of the benchmarks as well as the compilers used for the experiment. The last section describes the actual test plan which will be used to evaluate the potential effects of de-optimization on the optimization process.

2.1 VPO Interactive System for Tuning Applications (VISTA)

In order to investigate the potential benefits of de-optimization with respect to re-sequencing optimization phases, it was necessary to modify the VISTA (VPO Interactive System for Tuning Applications) compiler framework [22, 15]. VISTA was developed as an interactive compiler that would allow a knowledgeable programmer to finely tune the optimization phase order performed on a given function. To facilitate this, VISTA provides a graphical interface that interacts with a VPO (Very Portable Optimizer) backend. Using this GUI, a programmer can tailor the phase sequence based on immediate performance feedback information generated using EASE (Environment for Architecture Study and Experimentation). However the most attractive feature of VISTA is the ability to automatically search for effective phase sequences using a genetic algorithm. This is particularly beneficial in the case of attempting to reapply transformations on de-optimized assembly code, since we are assuming that the original phase ordering may have been suboptimal.

2.1.1 Very Portable Optimizer (VPO)

A fundamental component of VISTA is VPO, the actual compiler backend with which it communicates. VPO [2] was designed to be both portable and efficient, accepting an intermediate language known as RTLs (Register Transfer Lists), and producing assembly code. These RTLs provide a machine-independent representation of the effects of machine instructions. By operating on these RTLs, various code-improving transformations found in VPO can be written in a machine-independent manner (e.g. loop-invariant code motion), with only a small portion of the code requiring any information about the specifications of the target machine. Thus the amount of work necessary for porting VPO to a new architecture is reduced.

VPO also incorporates EASE (Environment for Architecture Study and Experimentation) [7] for instrumenting generated assembly files to collect performance data. Both static information (code size) and dynamic information (instructions executed, memory references) can be collected through the use of this profile data. VISTA enables the programmer to query these frequency measures at any point during the compilation process, and thus facilitates fine tuning of the code.

2.1.2 Searching for Effective Phase Sequences

Due to the nature of embedded application development, longer compile times are often tolerated to allow further improvements to the generated code, since code size and timing requirements are typically very inflexible. Evolutionary algorithms are effective in dealing with large search spaces for which the interaction between parameters is not well understood. Since the phase ordering problem has been characterized in this manner [6], a genetic algorithm approximation is a good solution. The VISTA framework provides such a genetic algorithm search for effective phase sequences [13, 15, 14].

Figure 2.1 depicts the genetic algorithm search window with various parameters that are able to be adjusted by the programmer. The parameters include the optimization phases available to the algorithm, the maximum number of phases to perform, the population size, the number of generations, the type of search to perform, as well the fitness criteria to use in evaluating the phase sequences. When selecting the fitness criteria, the programmer

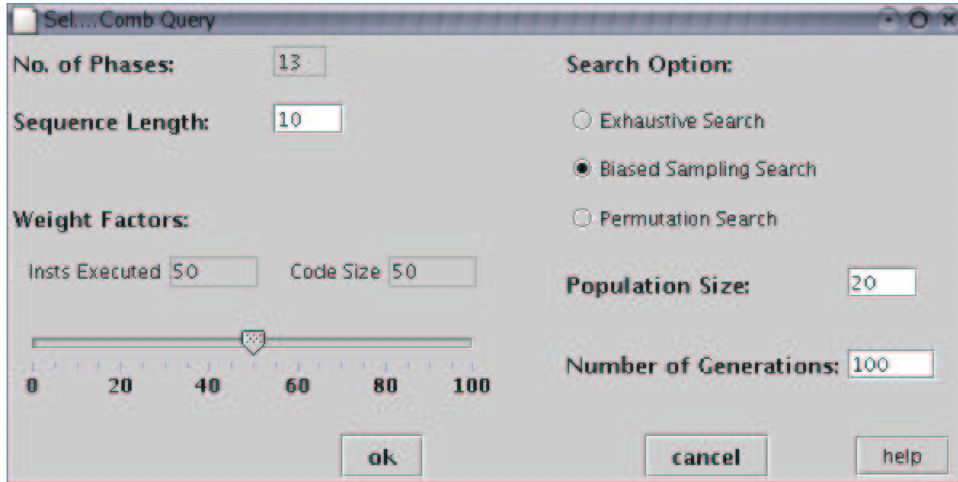


Figure 2.1: VISTA Genetic Algorithm Search Parameters

can choose between optimizing for code size, instructions executed, or a mixture of both. Optimization phase sequences are then tested systematically by the genetic algorithm to locate the most beneficial ordering it can find based on the fitness criteria selected.

The genetic algorithm can also be prematurely stopped by the programmer if the desired level of optimizations are achieved before completing the entire specified search. Figure 2.2 shows a search in progress after 448 attempted sequences. Note that invalid sequences are displayed and recorded to assist in the debugging of new optimizations and features added to VISTA. Both the current sequence being tested and the best found sequence are displayed using simple letters to represent the various optimization phases. Statistics about the performance of the best sequence can be found at the bottom of the window. Sequence 317 produces code that is 47% of the original code size as well as only executing 48.4% of its original instructions. In this example, the fitness criteria is mixed 50%/50% for static and dynamic count measures, thus showing a 47.7% relative fitness for the best sequence found in the search to this point.

2.2 Hardware and Software Platform

The proposed modifications to VISTA were tested using an Intel StrongARM SA-110 processor running Netwinder Linux. Due to the low clock speed of the StrongARM processor, several other machines were used for cross-compiling for static code size only if the function

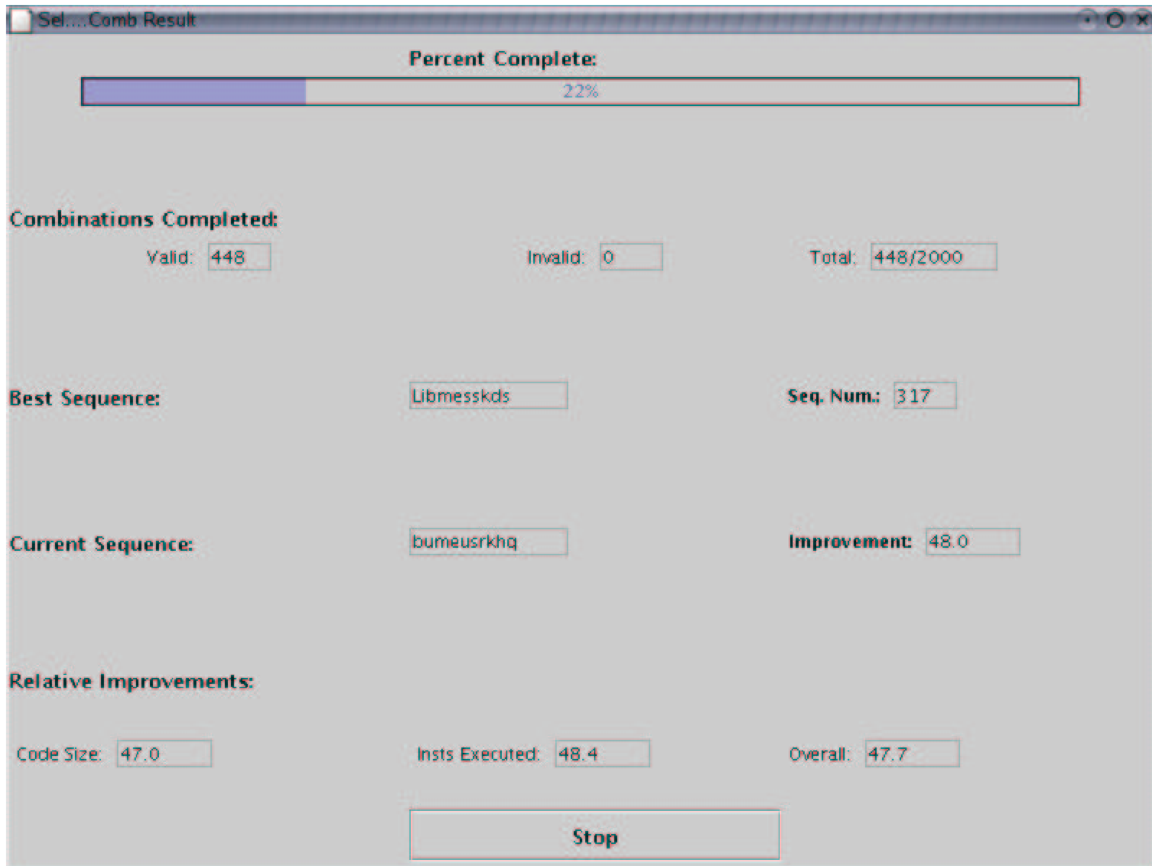


Figure 2.2: VISTA Genetic Algorithm Search (In Progress)

being optimized was not executed by an application given the sample test data. In such cases, only the static code size measure could be extracted and thus used to guide VISTA in the automatic tuning of the code. VISTA typically uses the dynamic instruction count as a tie-breaker for cases where the function is executed by the application. The machines used for cross-compiling included a Sun Ultra SPARC III as well as an Athlon XP.

Due to the proprietary nature of most embedded applications, it is quite a challenge to find hand-generated assembly programs that are representative of the typical workload. To the best of our knowledge, there are no currently available embedded benchmark suites written in assembly code. Rather than devising a new set of hand-tuned assembly benchmarks for embedded systems, an optimizing compiler could be used instead with a comparable existing high-level source code benchmark suite. Although using actual hand-tuned assembly code would be better, the use of known optimization techniques with

a known benchmark suite provides a legitimate testing framework for evaluating the effects of de-optimization when re-optimizing code.

2.2.1 MiBench Embedded Applications Benchmark Suite

In order to effectively gauge the effectiveness of performing de-optimizations on optimized assembly, we selected an appropriate benchmark suite. The applications available in the MiBench embedded applications benchmark suite are described as representative of common programs used in embedded systems [9]. The suite consists of several programs from the following six categories: Automotive/Industrial, Network, Telecommunications, Consumer, Security, and Office. For the experiments performed, one application was selected from each area as shown in Table 2.1.

Table 2.1. MiBench Benchmarks Used for Experiments

Category	Program	Description
Automotive/Industrial	bitcount	Bit manipulation tests
Network	dijkstra	Dijkstra's shortest path algorithm
Telecomm	fft	Computes a Fast Fourier Transform
Consumer	jpeg	Creates a jpeg image from a ppm
Security	sha	NIST Secure Hash Algorithm
Office	stringsearch	String pattern matcher

Three benchmark programs required minor changes in order to translate and compile correctly using the modified version of VISTA. In the benchmark *stringsearch*, a large local array was changed to static in order to not require additional information to be supplied to the translator. This change does not impact the experimental comparison since neither GCC nor VISTA modify the placement of local arrays on the stack. In the *fft* benchmark, several functions were adjusted to pass a pointer to a data structure containing floating-point arguments instead of each individual argument. This change was necessary as calling conventions for the ARM are complex when dealing with floating-point arguments that get passed in registers and on the stack. This further complicates the process of translating these parameters appropriately. Performing this fix will inhibit performance slightly as an additional pointer dereference is now required to access each argument. This fix helps

during translation to reduce the amount of outside information necessary for correct behavior. Additionally, for the *jpeg* benchmark, the function *parse_switches* was not able to be tested using VISTA. This was due to a problem with code expansion during the translation pushing global variable references further from the actual global variable locations. Further discussion of assembly code requirements for translation and information loss is available in Chapter 3 and Chapter 4.

2.2.2 Generating Assembly Code

In order to allow a fair test of the the proposed de-optimization strategy, it was necessary to run each program through an optimizing compiler. Each of the programs from the MiBench suite was compiled and optimized using the GNU Compiler Collection's C compiler version 3.3 [8] for the ARM. The exact command line used to compile each of the C source files was as follows:

```
gcc -O2 -S -c -fno-optimize-sibling-calls -ffixed-lr -ffixed-fp filename.c
```

This command line can be broken down as shown in Table 2.2. The options specified are necessary to provide for as little information loss as possible when translating from the assembly. Level 2 optimization allows a fair comparison to hand-generated assembly since it does not invoke additional phases that will contribute to increased space requirements. The *-S* and *-c* flags force GCC to generate assembly code as output. Translating from object code is possible, but would add an additional unnecessary complexity to the translation process for our purposes.

The *-f* flags that were selected disable specific phases of the *-O2* optimization process. Optimizing sibling calls is a transformation that allows leaf or sibling functions to omit save and restore instructions. Allowing GCC to perform these optimizations makes it harder for the translator to analyze where the function exits. Additionally VISTA will automatically re-perform this translation on the code both for the GCC baseline code as well as the experimental code. The two *-ffixed* flags force GCC to disallow the use of these registers as general purpose registers. This is necessary to perform a fair comparison since VISTA currently does not support using either the Frame Pointer or the Link Register in a different manner than that for which they were intended.

Table 2.2. GCC Optimization Flags

Flag	Meaning
-O2	Level 2 optimizations (Standard)
-S	Generate assembly file but do not assemble
-c	Compile and do not link
-fno-optimize-sibling-calls	Do not perform optimizations for leaf or sibling functions
-ffixed-lr	Do not use the Link Register (LR) as a general purpose register
-ffixed-fp	Do not use the Frame Pointer (FP) as a general purpose register

2.3 Experimental Test Plan

Using the MiBench benchmark programs, each GCC-generated assembly file is translated to RTL format using the ASM2RTL translator as discussed in Chapter 3. These input files are then compiled using VISTA and tested for correctness. These results form the baseline for our further measurements. After verifying the newly translated code, each function is tested for immediate code improvements from the traditional optimization phases performed by VISTA. Additional optimization phase orderings are selected using the genetic algorithm search with three different fitness criteria: 50% code size/50% dynamic instruction count, 100% code size, and 100% dynamic instruction count. Next the de-optimizations are turned on before applying further VISTA optimization phases, and data is again collected using the three fitness criteria. For the de-optimizations, first *loop-invariant code motion* is undone, followed by the de-optimization of *register allocation*. Further details of the entire de-optimization process are described in Chapter 4. For each of the test cases, 14 optimization phases are available from which the genetic algorithm can choose. These phases are described in greater detail in the top portion of Table 2.3. The bottom portion of the table displays two required phases for the genetic algorithm. *Register assignment* is performed after *register allocation* is de-optimized and *fix entry exit* is performed as the final phase for each function compiled.

From the data collected, comparisons will be made between code size and dynamic instruction count of both the optimized code and the de-optimized plus re-optimized code

for each benchmark. The potential benefit of assembly translation and further optimization can be shown between the GCC-generated code and the results of optimizing that code. Potential benefits of de-optimization plus re-optimization will be shown by comparing with the results of the optimized VISTA translated assembly code.

Table 2.3. VISTA Genetic Algorithm Candidate and Required Phases

Optimization Phase	Description
Branch Chaining	Replaces a branch or jump target with the target of the last jump in a jump chain.
Common Subexpression Elimination	Eliminates fully redundant calculations which also includes constant and copy propagation.
Remove Unreachable Code	Removes basic blocks that cannot be reached from the entry block of the function.
Remove Useless Blocks	Removes empty basic blocks from the control flow graph.
Dead Assignment Elimination	Removes assignments when the assigned value is never used.
Block Reordering	Removes a jump by reordering basic blocks when the jump target has only a single predecessor.
Minimize Loop Jumps	Removes a jump associated with a loop by duplicating a portion of the loop.
Register Allocation	Replaces references to a variable within a specific live range with a register.
Loop Transformations	Performs loop-invariant code motion, induction variable elimination, and loop strength reduction on each loop, ordered by loop nesting level.
Merge Basic Blocks	Merges two consecutive basic blocks a and b when a is only followed by b and b is only preceded by a.
Strength Reduction	Replaces an expensive operation with one or more cheaper ones.
Reverse Jumps	Eliminates an unconditional jump by reversing a conditional branch when it branches over the jump.
Instruction Selection	Combine instructions together and perform constant folding when the combined effect is a legal instruction.
Remove Useless Jumps	Removes jumps and branches whose target is the following block.
Register Assignment	Maps all pseudo-registers to hardware registers and generates any necessary spill code
Fix Entry Exit	Arranges remaining local variables on the stack and generates all necessary save and restore code for function entry and exit points. Performs predication of RTLs if possible.

CHAPTER 3

DEVELOPMENT OF ASM2RTL TOOLS

This chapter focuses on the translation of assembly source code to the VISTA RTL format, and the ASM2RTL tool suite that was developed to facilitate this process. The first section covers the basics of assembly translation and re-optimization as well as why this can be such an attractive option for embedded applications development. Next we explain some of the problems surrounding the translation process itself, specifically with regard to maintaining program correctness. The next section discusses some implementation strategies to make the tool suite as robust as possible. Difficulties encountered during the translation process as well as the chosen solutions are then presented in the following section. The final section describes additional translators that were developed for use with the ASM2RTL tool suite, but not used for the purposes of this study.

3.1 Assembly Translation and Re-optimization

The process of assembly translation discussed in this thesis refers to the conversion of assembly source code to an intermediate language for use with an optimizing compiler. Translation enables an iterative process for re-optimizing programs with potentially different compilers and optimization techniques. SALTO is a tool for assembly transformation and optimization that is employed in such an iterative process [18]. Low-level optimizations are applied to assembly code by SALTO as part of an iterative sequence including both a high-level restructurer and an optimizing compiler. However, only information about the transformations performed by SALTO are passed back to the original optimizing compiler for the next pass. In this way, no translation is performed to convert the assembly code back to the intermediate language. This is slightly different from the proposed strategy for re-optimization presented in this thesis, where optimized assembly code is actually translated into the appropriate intermediate form.

Figure 3.1 shows one potential view of an iterative process for the further optimization of assembly source code. One of the main advantages of this arrangement is the ability to use various independent optimizing compilers for each stage of the optimization, so long as an appropriate translator has been constructed for its corresponding intermediate language. A high-level source file can be compiled with a traditional compiler or a hand-tuned assembly file can be used as the initial input to the system. Assembly code is then translated to an intermediate language format and fed to a corresponding optimizing compiler. This process can be be repeated with additional translators and optimizing compilers as many times as necessary. Additional passes could also be substituted with hand-tuning of the assembly code by a programmer. It is easy to see that this scheme can be further extended to incorporate profiling data with each optimizing compiler pass, exposing even further opportunities for fine-tuning the code.

Although performing optimization passes in an iterative manner will be more time-consuming, the benefit may be invaluable, particularly for applications with strict requirements. Embedded applications can have real-time considerations or code size limitations that must be rigidly adhered to. These types of constraints make complex optimization strategies such as an iterative model attractive since the increased optimization benefits can outweigh the additional overhead cost.

This thesis focuses on the effects and possible benefits of reordering optimization phases via de-optimization and re-optimization. To study this, it was necessary to construct a translator from native ARM assembly code to VISTA RTLs (Register Transfer Lists). This translator is part of a larger suite of ASM2RTL tools, a group of translators in which each converts instructions from a given assembly language to RTLs. The current version of ASM2RTL supports assembly instructions from the Sun Ultra SPARC III, the Texas Instruments TMS320c54x and the Intel StrongARM. Assembly translation to the intermediate RTL form which VISTA accepts appears to be mostly straightforward, but it does contain several potential pitfalls.

Almost every problem that can occur is due to the loss of semantic content needed for correct operation. However there are other potential problems in translation, such as the lack of support from the VPO compiler for a very context-specific instruction (e.g. predicated return instruction). If a compiler cannot produce a particular instruction, it must be replaced

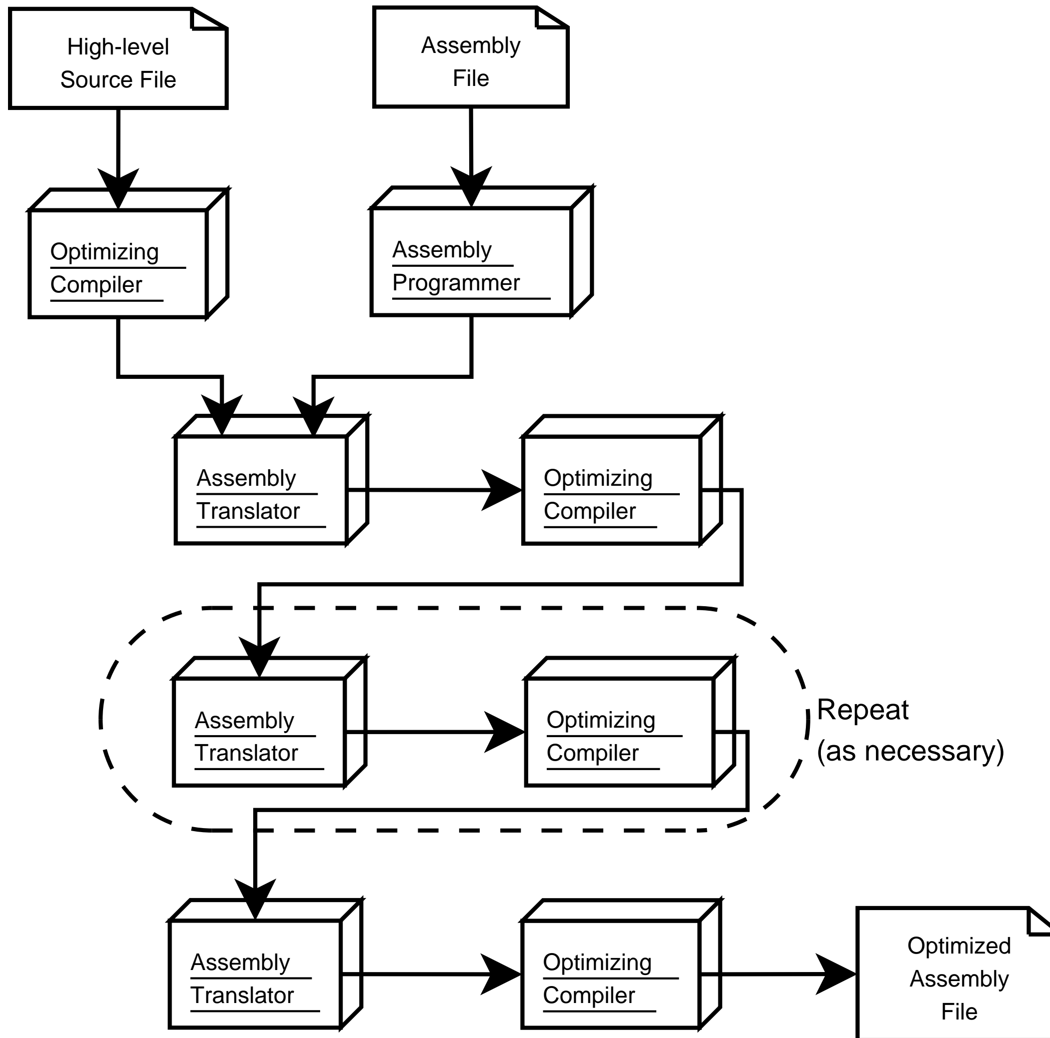


Figure 3.1: Iterative Re-optimization of Assembly Code

by a semantically equivalent sequence of instructions that preserve all other aspects of the current program state.

3.2 Preservation of Program Semantics

3.2.1 Information Loss

It is well known that translating from a high level language to assembly language is easier than translating from assembly language back to a high level language. The information contained within a program can be characterized as flowing from its highest level source

code down to the final machine code. As a program progresses through various intermediate representations, each lower level form carries less information than preceding higher level forms. Thus it becomes increasingly difficult (if not impossible) to extract the entire original program representation from a low level form. When attempting to reconstruct original semantic content for a compiled program, one problematic area is the detection and characterization of local variables. Another problem is maintaining consistency with the calling conventions of the given instruction set architecture (ISA).

3.2.2 Local Variables

Local variables (or automatic variables) are those which are kept on the run-time stack. Modern ISAs support memory accesses of fixed size increments. As an example the StrongARM supports memory access sizes of 1 byte, 2 bytes, 4 bytes, and 8 bytes. However data that is larger than these sizes can only be accessed by breaking it into pieces that conform with these sizes. Arrays and structures are typically larger than these fixed sizes and are thus handled by moving the necessary pieces into registers using these standard memory-accessing instructions. From a low-level assembly representation of such a program, it is difficult to distinguish between scalar and array data.

This is problematic when doing translation and re-optimization, since the VISTA RTL representation (like most intermediate languages) handles local variables symbolically. This means that the original numeric offset information is lost during the translation, and the ordering of local variables on the stack can change. This reordering can cause the generated code to be semantically incorrect, since local variables larger than 4 bytes may be split apart and spread out on the stack. This is especially true for local structures where certain fields may be manipulated while others are ignored. Calling a function using a pointer to such a split structure would cause other local variables to be incorrectly read and written in memory, since the C language assumes each function uses the same structure layout. The same argument holds for arrays as well, since they are merely a constrained structure where each field is of the same data type.

Figure 3.2 shows an example where optimized ARM assembly code can easily be misinterpreted. The corresponding RTLs are shown as comments to the right of each instruction. Both code snippets come from the function *fft_float()* in the FFT benchmark.

Double placed in two integer registers		
Line	Instruction	RTLs and Comments
	...	
1	add r2, sp, #8	r[2]=r[11]+LOC[8];
2	ldmia r2, {r2-r3}	r[2]=R[r[2]];r[3]=R[r[2]+4];
3	str r2, [r5, #16]	R[r[5]+16]=r[2];
4	str r3, [r5, #20]	R[r[5]+20]=r[3];
	...	

Two integer loads coalesced in same manner		
Line	Instruction	RTLs and Comments
	...	
1	add r2, sp, #40	r[2]=r[11]+LOC[40];
2	ldmia r2, {r2, r3}	r[2]=R[r[2]];r[3]=R[r[2]+4];
3	add r3, r3, r2	r[3]=r[3]+r[2];
	...	

Figure 3.2: Local Variable Confusion

The top selection of code shows the movement of a double from memory (using the *ldmia* instruction) to two integer registers. This is typically performed for function calls which pass up to the first four parameters (including floating point values) in integer registers. In order to generate correct code, the local variable at location *sp+8* must always be considered a double and should not be allowed to be split into two 4-byte integers by VISTA. The bottom portion of code starts off in a similar fashion, loading two registers using a single stack offset with a multiple load instruction. These two registers are then added together, clearly something that is not typical for two 4-byte halves of a double. However it could also be possible that this is part of an integer array where the first two elements are added together (potentially as in a hashing algorithm). As it turns out, these two local variables are indeed just two distinct integers, but that conclusion could not be made without having some additional information provided alongside the assembly code.

3.2.3 Calling Conventions

Another important factor in performing the translation of assembly code to VISTA RTL format is maintaining proper calling conventions. The RTL format requires that registers

used as function arguments as well as the size of the arguments passed on the stack be specified explicitly. The registers containing return values must also be specified explicitly. The StrongARM calling conventions can be summarized as follows:

1. As many as 4 arguments to a function may be placed in registers using the following order: r0, r1, r2, r3. Additional arguments (beyond 4 words) to functions are to be placed on the runtime stack.

2. Floating point arguments will be placed starting in integer registers, and will be split (e.g. r3 and the first argument on the stack) if necessary. The same holds true for structures.

3. Variable argument length functions place data in registers similarly, and all additional arguments are pushed onto the runtime stack in reverse order.

4. Integer returns are handled using r0. Floating point data is returned via f0. Data structures smaller than 4 words can be returned using as many registers from r0-r3 as necessary. Data larger than this size is handled via an additional address parameter to a structure.

The important considerations here are to make sure that no information is lost during the translation process. If registers that are used as arguments are not specified, then it is possible that VISTA will detect sets to these registers as dead assignments and thus eliminate them. Additionally if registers that contain return values from function calls are not specified appropriately, VISTA will generate spill code around the function call to handle reads from these registers. In each particular case, this will cause incorrect code to be generated for the function, and thus needs to be handled carefully.

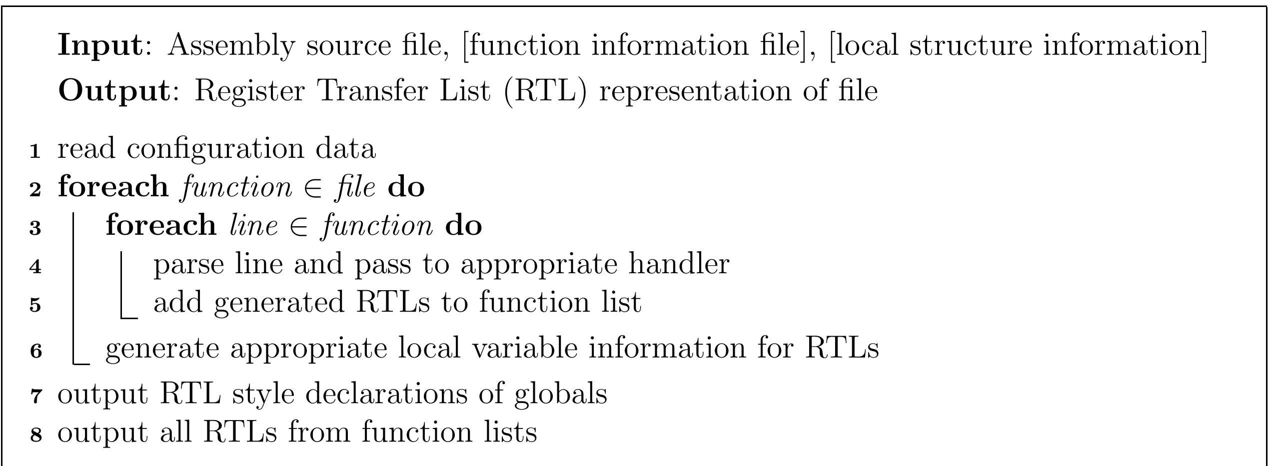
3.3 Implementation Strategy

The ASM2RTL translator suite was implemented to facilitate retargetability for the source assembly language. Thus there is a large portion of code which is machine-independent. The machine-dependent portions of the code are relatively easy to construct, requiring the modification of several subroutines to perform activities correctly such as

operand parsing or assembly instruction identification. The simplified program structure of the translator is displayed in Figure 3.3. Each phase of the translation process will be discussed, but not necessarily in the order in which they took place.

To make the translation process as fast as possible, lines were translated individually in the file, maintaining only a small amount of state information concerning things such as local variables, global variables, function return types and argument counts. Once the instructions were converted to RTL file format, VISTA would be able to reconstruct any additional information it needed by performing the standard analysis phases that it would normally perform with any other supplied input file.

The ASM2RTL translator begins by reading in all necessary configuration information (Line 1). This required information is discussed further in Section 3.4. Using the ASM2RTL translator, each line of the assembly file is parsed and translated into a semantically equivalent sequence of RTLs (Lines 3-5). These RTLs are maintained in a list for output at the end of the program (Line 8). Typically the correspondence is one-to-one, but whenever a global variable or local variable is first encountered, additional RTLs for declarations are also generated. Additionally there are some instructions which are not representable via a single RTL currently and thus the translator will produce an equivalent sequence of instructions instead. Functions are handled one at a time (Line 2), and a post-pass is performed on each function before proceeding to translate the next function (Line 6). After translating each line in the input file, the translator will then proceed to generate the RTL file by emitting



Input: Assembly source file, [function information file], [local structure information]
Output: Register Transfer List (RTL) representation of file

1 read configuration data
2 **foreach** *function* ∈ *file* **do**
3   **foreach** *line* ∈ *function* **do**
4   parse line and pass to appropriate handler
5   add generated RTLs to function list
6   generate appropriate local variable information for RTLs
7 output RTL style declarations of globals
8 output all RTLs from function lists

Figure 3.3: ASM2RTL Program Structure

all necessary global declarations first (Line 7) and then finally each of the function's RTL lists in order (Line 8).

3.4 Translation Difficulties

The two major problems faced when performing translation of assembly code to the VISTA RTL format are maintaining memory consistency and abiding by the calling conventions of the target environment. Both of these problems occur due to the loss of information as the program is converted from a high level source language to assembly language. In some cases, no additional information is necessary to be supplied to the translator, as ASM2RTL can use a few simple techniques to recover such information from the underlying code structure of the assembly file. However, some assembly code is not able to be translated correctly without supplemental information.

3.4.1 Memory Consistency

Maintaining proper memory layout information for local structures is vital to obtaining correct code with VISTA. If structures, arrays or even the two 4-byte halves of a double are allowed to be reorganized with VISTA, then the resulting code will be semantically incorrect. Additionally, memory errors such as segmentation faults and buffer overflows will occur when running the new code. Thus to protect against these types of translation errors, additional information concerning the locations and sizes of local variables is necessary.

This is handled in ASM2RTL by supplying an additional command line parameter along with a file containing function names, followed by lists of structure location, structure size pairs. If specified, this file is read in by ASM2RTL on startup and an additional post-pass for fixing structure locations is invoked before completing the code translation for a particular function.

The method used for producing correct memory layout information is shown in Figure 3.4. First all local variable references that relate to function arguments (either incoming parameters or outgoing arguments) are fixed as static offsets, similarly to the code that the standard VISTA frontend compilers generate (Line 1). Next we perform the structure modifications if they were indeed specified by the user (Line 2). We compare each structure in the list

with each local variable offset we have seen (Lines 3-5). If a local is found to be part of a larger data structure, we immediately calculate a new offset for it and replace all references to the original local variable with the structure location plus new offset in the function's RTLs (Line 6). Since all references to this symbolic local variable have been replaced, we can now safely remove it from the local variable symbol table (Line 7).

```
1  fix all function arguments on the stack as unmodifiable
2  if fix_structs_flag then
3  |   foreach struct ∈ structs_list do
4  |   |   foreach var ∈ local_vars do
5  |   |   |   if struct.loc < var < (struct.loc + struct.size) then
6  |   |   |   |   replace var in RTLs with struct.loc + #(var - struct.loc)
7  |   |   |   └   remove var from local_vars
8  extract locs from local symbol table
9  sort locs by offset in increasing order
10 foreach var ∈ locs do
11 └   var.size = nextvar.offset - var.offset
12 prepend local variable declaration RTLs to function list
```

Figure 3.4: Local Variable Reconstruction

After fixing all local structure references appropriately, the algorithm proceeds to a local variable size auto-detection phase. Each local variable is taken from the symbol table and sorted in increasing order by offset (Lines 8-9). This list can then be traversed in order to access the next neighbor for each particular local offset. Using this method, the size of a local can be calculated as the difference between its next neighbor's offset and its own offset (Lines 10-11). Since we have the total stack size from an initial stack pointer save instruction at the start of the function, we also know the size of the last local variable on the stack. Finally, each local variable has a declaration RTL constructed for it. This RTL is prepended to the current function's list of RTLs (Line 12).

Due to the last phase of reconstruction (Lines 10-11), any array or structure that only has its starting address taken will require no additional information to be supplied in the structs file. This is beneficial in that it reduces the amount of inspection a user must do for the incoming assembly code to detect such memory layout problems. Since array elements

are rarely accessed directly by offset (e.g. arr[5]), and structures tend to be dynamically allocated in the heap, the need for user-supplied structure information is minimal.

3.4.2 Following Calling Conventions

A typical RTL input file contains extra information regarding registers and memory locations that are used for special purposes relating to the calling conventions of the StrongARM. Examples of such meta-information RTLs are USELINEs (implicit uses of registers), SETLINEs (implicit sets of registers), PARMLINEs (parameters to a function), CSLINEs (caller save registers), and RESLINEs (registers that are reserved). In order to produce these RTLs properly, it was necessary to detect incoming register arguments, incoming stack-placed arguments, outgoing register arguments, outgoing stack-placed arguments, as well as registers containing return values. It is possible to argue that this data is unnecessary, since all necessary arguments and return values have been set up correctly in the incoming assembly file. This approach could work if VISTA was only to be used for converting the RTLs directly to assembly instructions, however various analyses could be miscalculated without this information and thus code-improving transformations might eliminate necessary instructions.

Live register and variable analysis could be used to detect some incoming parameters and outgoing parameters in a function. To perform live register and variable analysis, the entire control flow graph of the program needs to be constructed. Even at this point, some of the information, such as the size of incoming stack arguments, would still be unavailable as it was lost already at a higher level. Since performing such inter-procedural analysis is time-consuming and may not even yield entirely correct information in these cases, ASM2RTL was set up to strictly perform line-by-line translation. Thus information about live registers and variables for functions needed to be supplied in some other manner than the original input assembly file. To remedy this problem, we allow ASM2RTL to read in configuration information about functions used by the application being translated.

ASM2RTL scans for configuration information about functions when started. Simple text files are parsed for information about function return types and incoming argument sizes in 32-bit words (the smallest unit of allocation for the StrongARM). Data about functions was split into two files, one for globally accessible functions (library or system calls), and one

for locally accessible functions (functions defined by the application). This information can then be used during the translation process to reconstruct the appropriate meta-information in RTL format. As an example, knowing the return type of a function allows ASM2RTL to generate appropriate RTLs for maintaining this data as live when exiting the function, so that an RTL updating the return value register is not inappropriately deleted when performing *dead assignment elimination*. Local configuration files can be created by either inspecting the assembly code and interpreting the necessary information, or easily extracted from the original high level source code (if available). The global function configuration file was easily created using library and system call information found in references for the standard C libraries.

Variable length argument functions such as *printf*() prove to be problematic when translating as well. Because the number of arguments can exceed the number of argument registers, the remaining arguments placed on the stack cannot be reorganized. Thus it is necessary for ASM2RTL to know the exact number of variable parameters used for each call. This is another example of the information lost during the compilation process, so it must be resupplied to the translator. To correct for this, a simple tool was constructed to search for common variable length argument functions in the initial C source files. Using this tool, each variable argument function's name was suffixed with the size of its arguments. In this manner, the configuration file could specify *printf5*() as having 5 32-bit words of parameters.

Function pointers create another problem since the number of parameters to the actual function is only truly known in the initial source file. In the generated assembly file, the call instruction uses a single register for the actual function address. With ASM2RTL, all calls through function pointers are assumed to use 4 register parameters. Although this would invalidate some C code with more than 4 register parameters, this is not a very common situation, so the tradeoff of having to rewrite offending code is acceptable. Out of all 6 of the tested benchmark programs, there was but a single function that used function pointers, and it only used 3 register parameters.

3.4.3 Translation Tradeoffs

The ASM2RTL tool adopts various strategies for coping with problems that can affect proper translation. Each of these strategies is not without drawbacks. These strategies

require the programmer to inspect the supplied input assembly code and extract necessary information from it. There is a tradeoff involved since it is possible to assume a worst case scenario for each of the problems. In this way, no additional information is needed from the programmer, but code improvability is sacrificed.

With the local variable layout problem, one can assume that all elements on the stack belong to one large structure or array. In this case none of the elements are replaceable or reorderable. Doing this, however, will inhibit any further optimizations concerning these variables since arrays and structures are often ignored by the majority of code improving transformations.

In the case of following calling conventions, there are two requirements for guaranteeing consistency without additional information. First of all, the function must be assumed to be using its entire stack for argument space, and thus it is not able to be reorganized in any way. This is merely the same requirement as above for structures and arrays. Additionally, all argument registers and return registers must be marked as live using the meta-information RTLs in appropriate places. This will serve to inhibit many of VISTA's transformation phases from improving the code. One example of this would be the inability of *dead assignment elimination* to adequately detect dead assignments to argument or return registers.

3.5 Additional Translators

Development of the ASM2RTL tool suite started with the SPARC architecture, since the ISA is RISC-based and thus very orthogonal. Additionally, the SPARC port of VISTA has had the most extensive testing. The SPARC however was not chosen for this study, since we expected the largest gains to be due to undoing and re-performing *register allocation*. Preliminary tests showed that the MiBench benchmarks were just too small to create a significant amount of register pressure in the SPARC's 32 integer and 32 floating point register environment.

After completing the SPARC translator, a TMS320c54x-based translator was developed. Since its main use is as a Digital Signal Processor (DSP), the ISA is CISC-based and it has several addressing modes that make accurate decoding of instructions more difficult. Unfortunately, we found that the TMS320c54x-based port of VISTA was much less robust, so additional addressing modes were added to further facilitate the translation of assembly

instructions. Even so, several instructions that the native TI compiler exploited were not able to be realized within the VISTA environment. This led to significant decreases in performance, so a more appropriate embedded architecture was chosen for this study instead.

The StrongARM fits all criteria that we needed. It is an embedded processor having only 16 integer registers and 8 floating point registers. The ISA is very orthogonal with only a small number of instructions being hard to exploit by a compiler. Additionally the StrongARM port of VISTA is very robust and thus provided an excellent framework within which the proposed de-optimization strategy could be adequately evaluated.

CHAPTER 4

DEVELOPMENT OF DE-OPTIMIZATIONS

This chapter describes in detail the development and implementation of two new de-optimization phases to be added to the VISTA framework. Additionally the motivation for performing each de-optimization is also discussed. These de-optimizations were selected because each produces a noticeable effect depending on the phase order in which they are performed. The first de-optimization that is described deals with *loop-invariant code motion*. Next, the de-optimization of *register allocation* and *register assignment* is covered. Finally we discuss some of the implementation issues concerning the correctness of de-optimized code.

4.1 Loop-Invariant Code Motion

Loop-invariant code motion is a code-improving transformation that focuses on placing instructions outside of the loop, if they do not change the program state while in the loop. Such an instruction or RTL is called loop-invariant. A loop-invariant RTL is moved to the loop preheader. The preheader is a basic block that is the predecessor to the loop header, but is not in the loop itself. The preheader is only allowed to have the loop header as its successor. If more than one preheader exists for a loop, then a new preheader is constructed before the loop header. For an RTL to be considered loop-invariant, the following 4 conditions must hold true:

1. All source operands must be loop-invariant. This means that all operands are either immediate values, defined prior to entering the loop, or they are set by other loop-invariant RTLs.

2. The RTL must dominate all exits of the loop. This is to ensure that the RTL is executed on each iteration of the loop.

3. Any register that is set by the RTL cannot be set by any other RTL within the loop. A register that can change values is clearly not invariant.

4. Any register set by the RTL cannot be used prior to it being set by the RTL. Clearly a use before set in this case means that an RTL from outside the loop has set this register. Such a register can potentially be changing its value after an iteration of the loop causing the RTL to no longer be considered invariant.

VPO abides by each of these rules when performing *loop-invariant code motion*. In addition to attempting to move loop-invariant assignments to loop preheaders, VPO also attempts to place any loop-invariant expressions or memory references into registers, which can then potentially be accessed faster than the original reference. This approach requires and consumes additional registers for the purpose of allocation.

In the GCC-compiled code for the ARM, arrays typically have a starting address calculated and then later an offset is added to this address to access a particular array element. In a loop, the starting address calculation may be loop-invariant. Thus it can be moved to the loop preheader, thereby saving one dynamic instruction execution for each iteration of the loop. For deeply nested loops or those with high iteration counts, this can add up to a substantial savings in execution time.

4.1.1 Motivation for Undoing Loop-Invariant Code Motion

Loop-invariant code motion is a transformation that requires the use of additional registers to have the greatest benefit. These registers can be used to hold values such as loop-invariant variable loads, or complex arithmetic calculations that cannot be further simplified using traditional *strength reduction* or *instruction selection*. Increased register pressure in this case may inhibit additional code-improving transformations from being as beneficial as they possibly can be.

Undoing *loop-invariant code motion* provides VISTA with the possibility of applying additional code-improving transformations before potentially reapplying *loop-invariant code motion*. The re-insertion of loop-invariant instructions into loops will allow for other transformations to clean up the code, potentially reducing register pressure and exposing new opportunities for improvement. This provides a chance for suboptimal phase orderings

to be improved, where traditional optimizers would be unable to exploit any additional transformations due to existing constraints from prior phases.

The de-optimization of *loop-invariant code motion* can possibly be even more beneficial when applied to hand-generated assembly code. With hand-generated code, the testing of optimization phase sequences is orders of magnitude slower than with machine optimization. Thus it is even more necessary to rely on heuristics for knowing when to perform code motion and when not to. With loop constructs, it is very tempting to move all temporary values that are constant into registers outside of the loop, thus mimicking the effects of *loop-invariant code motion*. Such choices however could have terrible repercussions, as this naturally consumes a register that could otherwise be used for a different purpose. Careful visual inspection and testing can alleviate the overuse of code motion, but it is possible for more effective phase sequences to be masked due to what appears to be a good choice for *loop-invariant code motion*.

4.1.2 De-optimizing Loop-Invariant Code Motion

The strategy implemented for performing the de-optimization of *loop-invariant code motion* is shown in Figure 4.1. The algorithm focuses on placing loop-invariant RTLs back into loops before RTLs where a register they set is used. By setting these registers prior to any use within the loop, the initial loop-invariant RTL in the preheader can usually be removed from the program representation via *dead assignment elimination*. However if any set register is live on exiting the loop, the RTL cannot be safely removed at this time, and instead must remain intact. The nature of VISTA and its VPO backend relies on transformations such as *dead assignment elimination* and *instruction selection* to perform all unnecessary RTL removal, and thus the potential removal of a loop-invariant RTL from the preheader is postponed until such a phase is next performed.

This de-optimization algorithm can be broken down as follows: Loops are handled from the outermost to innermost (Line 1). This is in direct contrast with traditional *loop-invariant code motion*, where loops are handled starting from the most deeply nested. The loop is then subjected to loop-invariant analysis, to detect live register information in the context of the loop (Line 2). The de-optimization process begins by examining RTLs in the preheader of the loop in reverse order (Line 3). By working in reverse, loop-invariant analysis information

```
1 foreach loop ∈ loops sorted outermost to innermost do
2     perform loop_invariant_analysis() on loop
3     foreach rtl ∈ loop→preheader sorted last to first do
4         if rtl is invariant then
5             foreach blk ∈ loop→blocks do
6                 foreach trtl ∈ blk do
7                     if trtl uses a register set by rtl then
8                         insert a copy of rtl before trtl

9         update loop_invariant_analysis() data
```

Figure 4.1: De-optimize Loop-Invariant Code Motion

is kept up to date (Line 9), and loop-invariant RTLs are easier to detect and move into the loop. When a loop-invariant RTL is found in the preheader (Line 4), every basic block comprising the loop (Line 5) has its RTLs examined (Line 6). The algorithm then proceeds to insert a copy of the invariant RTL (Line 8) prior to any use of a register that it sets (Line 7).

The top portion of Figure 4.2 depicts a group of RTLs corresponding to a loop that has had *loop-invariant code motion* performed on it. In the example, the loading of a global variable containing the starting address of an array (Line a1) is a loop-invariant instruction that was moved out of the loop. The loop (Lines a3-a8) is performed 80 times using an induction variable that is set prior to beginning the loop (Line a2). Inside the loop, the loop-invariant register `r[10]` is used as part of an address calculation with the loop counter `r[6]`. The loop consists of only a single basic block (Lines a3-a8) and there exists a loop preheader (Lines a1-a2).

If the de-optimization procedure from Figure 4.1 is applied to this code, the following steps are performed. Loop-invariant analysis shows that the register `r[10]` is invariant for the loop, since it is live when the loop is entered, and it is not set anywhere inside of the loop. We then look at the last RTL in the loop preheader (Line a2). Examining this RTL shows us that it sets register `r[6]` which is not loop-invariant, and thus the instruction is skipped. Next we continue to scan backwards through the loop preheader and encounter another instruction (Line a1). This instruction sets register `r[10]`, which is loop-invariant. The source operand for this instruction is a memory location, which must be verified as invariant for the loop.

RTLs Before De-Optimization			
Line	Label	RTLs	Comments
		...	
a1		+r[10]=R[L44]	▷ Load a global variable (loop-invariant)
a2		+r[6]=0	▷ Initialize loop counter
a3	L11		
a4		+r[2]=r[10]+(r[6]{2)	▷ Calculate array address (global + counter)
a5		+r[5]=r[5]+R[r[2]]	▷ Add array value from memory to register
a6		+r[6]=r[6]+1	▷ Loop counter increment
a7		+c[0]=r[6]-79:0	▷ Set condition codes register
a8		+PC=c[0]'0,L11	▷ Perform loop 80 times
		...	

RTLs After De-Optimization			
Line	Label	RTLs	Comments
		...	
b1		**+r[10]=R[L44]**	▷ Load a global variable (loop-invariant)
b2		+r[6]=0	▷ Initialize loop counter
b3	L11		
b4		**+r[10]=R[L44]**	▷ Loop-invariant load (moved back into loop)
b5		+r[2]=**r[10]**+(r[6]{2)	▷ Calculate array address (global + counter)
b6		+r[5]=r[5]+R[r[2]]	▷ Add array value from memory to register
b7		+r[6]=r[6]+1	▷ Loop counter increment
b8		+c[0]=r[6]-79:0	▷ Set condition codes register
b9		+PC=c[0]'0,L11	▷ Perform loop 80 times
		...	

Figure 4.2: De-optimizing Loop-Invariant Code Motion

There are only memory reads inside the loop itself, and no proceeding RTL in the preheader modifies memory, so this memory reference is considered loop-invariant. Since the RTL now meets all the criteria for loop-invariance, it can safely be moved back into the loop by the de-optimization. Each RTL that makes up the loop is examined, and a use of **r[10]** is found (Line a4). A copy of the loop-invariant RTL is then inserted before this RTL that uses **r[10]** (before Line a4). The bottom portion of Figure 4.2 shows the loop after this invariant code has been moved back in. Even when the inserted RTLs do not dominate all loop exits, the original loop-invariant RTL (Line b1) will be removed by *dead assignment elimination*, provided that **r[10]** is not live leaving the loop.

4.2 Register Allocation and Register Assignment

Register allocation is a code-improving transformation that attempts to place local variable live ranges into registers because memory accesses are more expensive operations than register accesses. Moving local variable live ranges into registers also helps to enable additional code-improving transformations such as *instruction selection* and *common subexpression elimination*, which are more effective with register expressions. This process consumes registers as any *conflicts* with existing assigned register live ranges must be avoided.

Register allocation has been traditionally treated as a graph coloring problem, which can be defined as finding the minimum number of colors needed to color all vertices in a graph such that no two connected vertices share the same color. Graph coloring is considered NP-Complete, and as such an optimal solution is computationally expensive. Thus approximation algorithms are used instead. The application of graph coloring to *register allocation* treats registers as colors and live ranges as vertices. Live ranges that overlap or conflict with one another are connected by edges in the graph.

The graph used for performing *register allocation* is called an interference graph. Nodes are created for each local variable live range. Each node contains information about the local variable live range: sets, uses, allocated register (if any), a list of live registers, and a list of other nodes that conflict with the live range. Once the graph is constructed, nodes are colored by assigning an unused register for that particular live range. Since the number of registers available to the compiler is finite, the entire graph may not be color-able. It is also true that the allocation of non-scratch registers can add additional costs due to the necessity of saving and restoring their values on function entry and exit. Over-allocation is a problem that can be caused by allocating local variables that are only accessed infrequently. Critical choices must be made by the compiler as to which live ranges should be allocated and which potentially should never be allocated even if registers are available. Priority-based coloring is an approach that attempts to weight live ranges according to various heuristics so that a good solution can be obtained in a relatively short amount of time [4]. Similar approaches have been adopted by both GCC and VPO.

VPO traditionally receives input from a frontend that produces RTLs which does not make choices as to which registers should be used if possible. Instead the RTLs contain

references to pseudo-registers which do not actually exist. Certain registers, such as those used for parameter passing or return values must be specified by the frontend. *Register assignment* is the process by which pseudo-register references are then converted to actual hardware registers. The use of pseudo-registers facilitates certain phases in VPO such as *evaluation order determination* since choices for registers can be postponed until necessary. The process of *register assignment* is similar to *register allocation* in many ways, since conflicts of register live ranges must be avoided. In this case however, any pseudo-registers that cannot be allocated must have appropriate spill code generated.

4.2.1 Motivation for Undoing Register Allocation

Register allocation by definition requires the use of additional registers. This increased usage results in fewer registers available for other register-consuming transformations such as *loop-invariant code motion*. Similar to undoing *loop-invariant code motion*, the undoing of *register allocation* will also serve to decrease register pressure. The newly available registers can then be used with other potential transformations.

Whereas *loop-invariant code motion* can save cycles by not executing redundant instructions, *register allocation* enables local variable memory references to be converted to register references. Depending on the speed of memory accesses, this can be a huge gain, since functions tend to contain several local variables. However functions with many local variables are the problem, as choosing poor candidates for allocation can lead to decreased allocation opportunities for other local variable live ranges. Since other code-improving transformations can eliminate the need for register loads and stores, it makes sense that performing *register allocation* at different times in the phase sequence can yield varying results. With de-optimization, even temporaries that did not start the function as a local variable can be re-mapped using new local variable references (loads and stores). *Register allocation* can then be applied at a later stage to assign registers to the local variable live ranges that appear to give the greatest benefit.

The undoing of *register assignment* will potentially allow for even fewer registers to be used in various code sections, thus freeing up even more registers for other transformations. It is possible that the choices made during the initial run of *register assignment* require

additional stores and loads to preserve scratch registers around function calls. Changing the assignment could alleviate the need for such spill code entirely.

4.2.2 Register Interference Graphs

Figure 4.3 shows the procedure for constructing a register interference graph or RIG. The graph is constructed first by looking at each individual basic block (Lines 1-25). Later in the algorithm, the live ranges found in each basic block are connected together (Lines 27-34). The interference graph can be described as a collection of interconnected nodes. Each basic block in the function contains two new lists for the nodes, one called **`blkins`** which signifies the incoming register live ranges to the block and another list called **`blkouts`**, which represents the live ranges that are live on block exit. During the process of constructing the actual interference graph, **`blkouts`** contains the list of register live ranges for the current basic block.

Initially, each basic block inserts all valid incoming registers as nodes into **`blkins`** (Lines 2-3). After this is done, the **`blkins`** are copied into the **`blkouts`**, since all incoming registers are still live at this point (Line 4). Next, each RTL in the basic block is examined (Lines 5-21). Each RTL type is handled slightly differently. One RTL type is the parameter line, which signifies that the registers contained in the RTL are to be passed as parameters to a function. In this case, these registers should not be ever de-allocated since doing so would result in functions called with the wrong register parameters. Thus these RTLs are marked as not replaceable in the **`blkouts`** (Lines 6-7). With any other RTL, the registers used are examined and searched for in the **`blkouts`** (Lines 9-10). Any uses that are actually found have their node information updated, so that we can later revisit these RTLs (Line 11). If a matching live node is not found, then that means a return value from a function has been detected. In this case, a new node is created in **`blkouts`**, but it is marked unreplaceable (Lines 12-13).

Next, the RTL is checked for being a reserve line (Lines 14-15). Reserve lines are used to mark hardware registers as being already used. Typically these lines are associated with the assignment of function return values to different registers. Every RTL is then examined for registers that are set (Line 17). Each set register is then searched for in the **`blkouts`**, and if found, the node information is updated (Lines 18-19). Any register that is not found has a new node created and added to the **`blkouts`** (Lines 20-21). Finally the dead registers

```
1  foreach blk ∈ function basic blocks do
2      foreach reg live on entry to blk do
3          add new node to blk→blkins with rtl live register data
4      copy all blk→blkins into blk→blkouts
5      foreach rtl ∈ blk do
6          if rtl is a parameter line then
7              mark all parameter registers as not replaceable in blk→blkouts
8          else
9              foreach reg_used ∈ rtl do
10                 if reg_used ∈ blk→blkouts then
11                     update found node with rtl reg_used data
12                 else
13                     add new unreplaceable node to blk→blkouts with rtl reg_used data
14         if rtl is a reserve line then
15             add new node to blk→blkouts with return data
16         else
17             foreach reg_set ∈ rtl do
18                 if reg_set ∈ blk→blkouts then
19                     update found node with rtl reg_set data
20                 else
21                     add new node to blk→blkouts with rtl reg_set data
22         foreach reg_dead ∈ rtl do
23             search for node using reg_dead in blk→blkouts
24             update found node with rtl reg_dead data
25             remove node from blk→blkouts
26 foreach blk that can exit the function do mark all blk→blkouts as not replaceable
27 foreach blk ∈ function basic blocks do
28     foreach pblk ∈ blk→preds do
29         foreach node ∈ blk→blkins do
30             foreach pnode ∈ pblk→blkouts do
31                 if node and pnode use the same register then
32                     connect node and pnode as siblings
33                     if node or pnode are not replaceable then
34                         mark all siblings as not replaceable
```

Figure 4.3: Constructing a Register Interference Graph

associated with the RTL are examined and searched for within the `blkouts` (Lines 22-23). Sanity checks are also performed here, since no dead register can exist that is not currently live, thus existing in `blkouts`. When the proper node is located, its information is updated with the location of the dead register RTL (Line 24). Additionally, the node is removed from the current live range list known as `blkouts` (Line 25).

Any basic block that exits the function could potentially be returning a result to the calling function. Thus all live range nodes leaving such a block must be marked unreplaceable (Line 26). At this stage in the RIG construction process, all necessary nodes have been created and all that is left is the proper interconnection of live ranges. We proceed to connect live ranges of nodes by examining each basic block along with its predecessors (Lines 27-28). Each node live on entry to the basic block is compared with nodes leaving the predecessor's block (Lines 29-30). If both nodes work with the same register, then these nodes really correspond to a single register live range that spans multiple basic blocks (Line 31). In such a case, the two nodes are linked together in a circular list of siblings (Line 32). If either node is not replaceable (Line 33), then the entire list of siblings must be traversed to mark every node in the multi-block live range as irreplaceable (Line 34). This guarantees that unreplaceable nodes such as those belonging to incoming parameter registers are not inappropriately deallocated while they are still live, even if the basic block being examined is past the function entry block.

4.2.3 De-optimizing Register Allocation

The undoing of *register allocation* is analogous to the initial process of performing *register allocation*. In the case of undoing *register allocation*, however, the interference graph constructed is for register live ranges and not local variable live ranges. Any live range that can be de-allocated is assigned a new local variable which is loaded into a pseudo-register before any potential use. Additionally, any set is then followed by a store of that pseudo-register back to the new local variable location. It is also at this point that the pseudo-register is marked dead, so that each set or use requires individual store and load instructions. This serves to give the VPO optimizer the most flexibility when later re-performing *register allocation*.

The process of actually de-optimizing *register allocation* and *register assignment* is shown in Figure 4.4. Analysis phases for live variable information and dead register calculation are performed initially before attempting to de-optimize register allocation (Line 1-2). A register interference graph is then constructed via the procedure set forth in Figure 4.3 (Line 3). Each replaceable node in the RIG is then examined in turn (Lines 4-6). Nodes in the RIG can be either classified as *intrablock* or *interblock*. Intrablock nodes are those nodes which correspond to a live range that exists completely within a single basic block (Line 8). Alternatively, interblock nodes mark live ranges that span multiple basic blocks (Line 10). For the purposes of de-optimization, any intrablock live range that can be deallocated will only be assigned a pseudo-register (Line 9). Creating a new local variable for this case will not impact results since a local variable that is only live for a single basic block will almost always just be reallocated as a register. Interblock live ranges that are replaceable will have both a pseudo-register and a new local variable assigned to them (Lines 12-13). Each sibling node for this live range will also be updated with the local variable and pseudo-register information before being marked as done (Line 14). Nodes that are marked as not replaceable will not be assigned any pseudo-register or local variable.

Further analysis is then performed by VPO to handle the newly allocated pseudo-registers (Line 15). The second pass through the RIG will again examine each replaceable node in the RIG (Lines 16-18). Each intrablock node will replace any sets or uses of the original register with references to the newly allocated pseudo-register (Lines 20-22). The pseudo-register will then be marked as dead wherever the original register was marked dead. For intrablock live ranges, first the uses are replaced (Lines 26-28). Each use in an RTL will be replaced with the pseudo-register. The RTL will also have the pseudo-register marked as dead after this replacement. A new RTL must also be inserted before the RTL using the register, loading the appropriate pseudo-register with the allocated local variable. Next, each set is examined and replaced with a reference to the pseudo-register (Lines 29-30). An additional store instruction is inserted after the RTL, storing the pseudo-register's value back to its corresponding local variable (Line 31). The pseudo-register is then marked dead in the new store instruction RTL.

VISTA then re-performs *register assignment* at this point, minimizing the number of registers needed for the function to operate correctly (Line 32). A final call to *instruction*

```
1  calculate live variable information
2  calculate dead register information
3  RIG = construct_register_interference_graph()
4  mark all nodes in RIG as not done
5  foreach node ∈ RIG do
6      if ¬node→done ∧ node→can_replace then
7          node→done = TRUE
8          if node is an intrablock live range then
9              node→pseudo = new_pseudoregister()
10         else
11             ▷ node is an interblock live range
12             node→local = new_local_variable()
13             node→pseudo = new_pseudoregister()
14             update all siblings with local/pseudo and mark them done
15 recalculate necessary analysis for pseudo-registers in VPO
16 mark all nodes in RIG as not done
17 foreach node ∈ RIG do
18     if ¬node→done ∧ node→can_replace then
19         node→done = TRUE
20         if node is an intrablock live range then
21             for ref ∈ node→sets ∪ node→uses do
22                 replace ref with node→pseudo
23         else
24             ▷ node is an interblock live range
25             for sib ∈ node ∪ node→sibs do
26                 for use ∈ sib→uses do
27                     insert load of node→local into node→pseudo before use
28                     replace use with node→pseudo
29                 for set ∈ sib→sets do
30                     replace set with node→pseudo
31                     insert store of node→pseudo into node→local after set
32 re-perform register_assignment() to assign all pseudo-registers
33 perform instruction_selection() to clean up code
```

Figure 4.4: De-optimize Register Allocation

selection is then made to clean up redundant instructions that now appear due to the de-optimization process. Typically the resulting function representation will contain very few distinct registers. Most references will be to argument registers, reusable scratch registers and hardware-specific registers (frame pointer, stack pointer). Any unused registers are then available for the various code-improving transformations to use.

4.2.4 Example De-optimization of Register Allocation

This section walks through the steps of re-optimizing a function after de-optimizing *register allocation*. The initial code for the function being compiled is shown in Figure 4.5. This function is *dequeue* from the *dijkstra* MiBench benchmark. The *dequeue* function was chosen for its simplicity in demonstrating the benefits of de-optimization. Notice that the function has a NULL pointer check (Line 3) as well as a single call to the *free* library function (Line 8). The function takes three integer pointer parameters and returns void.

```
  void dequeue (int *piNode, int *piDist, int *piPrev)
  begin
1   static QITEM *qKill;
2   qKill = qHead;
3   if qHead then
4       *piNode = qHead→iNode;
5       *piDist = qHead→iDist;
6       *piPrev = qHead→iPrev;
7       qHead = qHead→qNext;
8       free(qKill);
9       g_qCount–;
  end
```

Figure 4.5: Dequeue from Dijkstra Benchmark

Figure 4.6 shows the starting RTL representation of the function after performing assembly translation on the GCC-generated assembly file. There are three basic blocks in the function, each separated by an additional horizontal line. Line 4 shows the check for the NULL pointer. Lines 6-8 show the saving of the argument registers `r[0]-r[2]` in non-scratch registers for later use. Line 10 ends the first basic block with a conditional branch to an exit block labeled L0001 if the *qHead* pointer is NULL. The second basic block

Line	RTLs	Deads	Comments
1	r[6]=R[L21];		
2	r[12]=R[r[6]+0];		
3	r[3]=R[L21+4];		
4	c[0]=r[12]-0:0;		▷ Check for NULL pointer
5	R[r[3]+0]=r[12];	r[3]	
6	r[4]=**r[1]**;	**r[1]**	▷ Saving argument **r[1]**
7	r[3]=**r[0]**;	**r[0]**	▷ Saving argument **r[0]**
8	r[5]=**r[2]**;	**r[2]**	▷ Saving argument **r[2]**
9	r[0]=r[12];		
10	PC=c[0]:0,L0001;	c[0]	▷ If NULL then goto L0001
11	r[2]=R[r[12]+0];		
12	R[r[3]+0]=r[2];	r[2]r[3]	
13	r[3]=R[r[12]+4];		
14	R[r[4]+0]=r[3];	r[3]r[4]	
15	r[2]=R[r[12]+8];		
16	r[1]=R[r[12]+12];	r[12]	
17	R[r[5]+0]=r[2];	r[2]r[5]	
18	R[r[6]+0]=r[1];	r[1]r[6]	
19	ST=free; =**r[0]**;		▷ Call free() with **r[0]**
20	r[2]=R[L21+8];		
21	r[3]=R[r[2]+0];		
22	r[3]=r[3]-1;		
23	R[r[2]+0]=r[3];	r[2]r[3]	
24	PC=RT;		▷ Return
25	L0001:		▷ Label
26	PC=RT;		▷ Return

Figure 4.6: Dequeue Prior To De-optimizing Register Allocation

performs the necessary pointer updates from the function. The call to *free* is found in Line 19, and it uses the register **r[0]** as its argument. Both lines 24 and 26 show return RTLs, signifying the end of the function *dequeue*.

After constructing the RIG and replacing hardware register references with pseudo-registers and memory operations, we obtain the RTLs shown in Figure 4.7. This figure only shows RTLs from the first basic block. If an RTL from the initial representation has been expanded into multiple RTLs from the de-optimization, the line numbers are suffixed alphabetically and the group of RTLs is separated out by horizontal lines. Lines 1a-1b show the new pseudo-register and memory version of the first original RTL. Notice that the

Line	RTLs	Deads	Comments
1a	**r[32]**=R[L21];		▷ **r[6]** → **r[32]**
1b	R[r[13]+**__dequeue_0**]=**r[32]**;	**r[32]**	▷ Store set of pseudo-register **r[32]**
2a	r[32]=R[r[13]+__dequeue_0];		▷ Load use of pseudo-register **r[32]**
2b	r[33]=R[r[32]+0];	r[32]	▷ Perform actual operation
2c	R[r[13]+__dequeue_1]=r[33];	r[33]	▷ Store set of pseudo-register **r[33]**
3	**r[34]**=R[L21+4];		▷ Intrablock live range so replace
			▷ with only pseudo-register **r[34]**
4a	r[33]=R[r[13]+__dequeue_1];		
4b	**c[0]**=r[33]-0:0;	r[33]	▷ **c[0]** is not replaceable
5a	r[33]=R[r[13]+__dequeue_1];		
5b	R[r[34]+0]=r[33];	r[33]**r[34]**	▷ Death of Intrablock **r[34]**
6a	r[35]=r[1];	r[1]	▷ **r[1]** is incoming argument
6b	R[r[13]+__dequeue_2]=r[35];	r[35]	▷ so not replaceable
7a	r[36]=r[0];	r[0]	▷ **r[0]** is incoming argument
7b	R[r[13]+__dequeue_3]=r[36];	r[36]	▷ so not replaceable
8a	r[37]=r[2];	r[2]	▷ **r[2]** is incoming argument
8b	R[r[13]+__dequeue_4]=r[37];	r[37]	▷ so not replaceable
9a	r[33]=R[r[13]+__dequeue_1];		▷ **r[0]** in this case is
9b	**r[0]**=r[33];	r[33]	▷ outgoing argument to free()
10	PC=c[0]:0,L0001;	c[0]	▷ Conditional branch uses only
	...		▷ **c[0]** so no replacements at all

Figure 4.7: Dequeue After De-optimization of Register Allocation

hardware register **r[6]** has now been re-mapped to pseudo-register **r[32]** for this particular live range. Since this live range exists across multiple basic blocks, its value is also assigned the new local variable **__dequeue_0**. The register **r[13]** corresponds to the stack pointer. It is verifiable that **r[6]** was still live going into the second basic block of the initial code example by looking at Line 18 from Figure 4.6. Notice that storing the pseudo-register also marks the pseudo-register as dead. The second RTL in the original function used the original **r[6]**, and so it is now loaded before the use in Line 2b. Using the same pseudo-register (in this case **r[32]**) allows VPO to keep the number of distinct pseudo-registers used in the function to a minimum. A new pseudo-register **r[33]** is chosen for the live range of **r[12]**. Line 2c contains the appropriate store instruction required due to the set of **r[33]** in Line 2b.

Intrablock live ranges were only assigned pseudo-registers and not new local variable locations. In Line 3, this is the case as **r[3]** is mapped to **r[34]** for its duration. This live

range of the original `r[3]` spans to Line 5 in the original RTLs, so `r[34]` is marked dead finally in Line 5b. With the ARM, it is not possible to map certain hardware registers to pseudo-registers. These hardware registers include argument registers, return registers, and the condition codes register `c[0]`. Lines 4b, 6b, 7b, 8b and 9b show that such register requirements are indeed respected in this function representation. Since the ARM only uses the register `c[0]` for conditional branches, instructions such as the one in Line 10 are never modified by the de-optimization algorithm.

The RTL representation of *dequeue* after re-performing *register assignment* is shown in Figure 4.8. Only the first basic block of the function is shown again. All pseudo-registers have been re-mapped to hardware registers, however the loads and stores will still remain present as *register allocation* has not been re-performed at this point in time. In Lines 1a-1b, the register `r[12]` is assigned to pseudo-register `r[32]` by VPO's register assignment procedure. Since this live range does not cross any call instructions (it only spans the 2 lines), VPO will assign the first available scratch register, `r[12]`. Since it is marked dead in Line 1b, it is available again for assignment in Line 2a. Not only is it assigned for the live range of `r[32]`, it is also assigned for `r[33]` since these ranges do not overlap each other. Of course `r[12]` is marked dead in Line 2c, where `r[33]` was previously marked dead due to the store inserted by the de-optimization process.

Lines 4a-4b show the use of a new hardware register, `r[3]` due to the existence of two live register ranges at this point in the basic block. The register `r[12]` is live due to its set as an intrablock live range in Line 3. Lines 6a-8b show the saving of incoming register arguments to new local variables. This function takes three integer pointers, and so registers `r[0]`, `r[1]`, and `r[2]` are saved appropriately. Since Line 9b sets `r[0]` as a parameter to the *free* function, it is still live exiting this basic block. The same is true for the stack pointer `r[13]` which will be live until the last reference to any local variable is seen.

The RTLs shown in Figure 4.9 are generated after performing several further optimizations including *dead code elimination*, *strength reduction*, *instruction selection*, *register allocation*, *common subexpression elimination*, *dead variable elimination*, and *fix entry exit*. Clearly all remaining local variable references have been eliminated due to the lack of any load or store instructions. This is consistent with the initial function. However the re-mapping of live register ranges has enabled additional transformations to be effective in reducing the

Line	RTLs	Deads	Comments
1a	r[12]=R[L21];		▷ **r[12]** is first non-argument
1b	R[r[13]+_dequeue_0]=r[12];	r[12]	▷ scratch register
2a	**r[12]**=R[r[13]+_dequeue_0];		▷ Note the use of **r[12]** to
2b	**r[12]**=R[**r[12]**+0];		▷ combine two distinct live
2c	R[r[13]+_dequeue_1]=**r[12]**;	**r[12]**	▷ ranges in these 3 lines
3	r[12]=R[L21+4];		
4a	**r[3]**=R[r[13]+_dequeue_1];		▷ First appearance of **r[3]** since
4b	c[0]=**r[3]**-0:0;	**r[3]**	▷ there are currently 2 live ranges
5a	r[3]=R[r[13]+_dequeue_1];		
5b	R[r[12]+0]=r[3];	r[3]r[12]	
6a	r[12]=r[1];	r[1]	▷ Save argument **r[1]**
6b	R[r[13]+_dequeue_2]=r[12];	r[12]	
7a	r[12]=r[0];	r[0]	▷ Save argument **r[0]**
7b	R[r[13]+_dequeue_3]=r[12];	r[12]	
8a	r[12]=r[2];	r[2]	▷ Save argument **r[2]**
8b	R[r[13]+_dequeue_4]=r[12];	r[12]	
9a	r[12]=R[r[13]+_dequeue_1];		
9b	r[0]=r[12];	r[12]	
10	PC=c[0]:0,L0001;	c[0]	▷ Live registers leaving block are
	...		▷ **r[0]** and **r[13]** (stack pointer)

Figure 4.8: Dequeue After Re-performing Register Assignment

number of RTLs. Originally the function required the saving of all three incoming register arguments. After re-performing *register assignment* the saves of registers **r[1]** and **r[2]** became redundant. Since the final use of **r[1]** and **r[2]** occur in Lines 14 and 17 respectively without any interceding function calls, these RTLs could use the registers directly. The missing save RTLs are evident as the two lines numbered 6 and 8.

4.3 De-optimization Difficulties

Several problems were encountered while trying to implement the various de-optimization strategies in a manner that preserved the semantic content of a function. Calling conventions posed a problem with the detection of registers that were able to be safely deallocated. Additionally, the expansion of code due to the insertion of new RTLs caused compatibility problems with global variable offsets specified in RTLs that were no longer syntactically valid in ARM assembly.

Line	RTLs	Deads	Comments
1	r[5]=R[L21];		
2	r[4]=R[r[5]];		
3	r[12]=R[L21+4];		
4	c[0]=r[4]:0;		
5	R[r[12]]=r[4];	r[12]	
6			▷ RTL **r[4]=r[1]** now unnecessary
7	r[8]=r[0];	r[0]	
8			▷ RTL **r[5]=r[2]** now unnecessary
9	r[0]=r[4];		
10	PC=c[0]:0,L0001;	c[0]	
11	r[12]=R[r[4]];		
12	R[r[8]]=r[12];	r[8]r[12]	
13	r[12]=R[r[4]+4];		
14	R[**r[1]**]=r[12];	**r[1]**r[12]	▷ **r[1]** live until here now
15	r[12]=R[r[4]+8];		
16	r[1]=R[r[4]+12];	r[4]	
17	R[**r[2]**]=r[12];	**r[2]**r[12]	▷ **r[2]** live until here now
18	R[r[5]]=r[1];	r[1]r[5]	
19	ST=free; =r[0];		
20	r[12]=R[L21+8];		
21	r[1]=R[r[12]];		
22	r[1]=r[1]-1;		
23	R[r[12]]=r[1];	r[1]r[12]	
24	PC=RT;		
25	L0001:		
26	PC=RT;		

Figure 4.9: Dequeue After Additional Optimizations

4.3.1 Calling Conventions Revisited

When implementing the de-optimize *register allocation* phase, calling conventions had to be maintained for proper function execution. This means that function arguments and return values need to remain in proper registers in order to interface properly with other compiled functions. This problem becomes a limiting factor for de-optimizations however, since hardware register references for such sequences must not be replaced with pseudo-registers or loads and stores. This limits the effectiveness of de-optimization in this case, since these references will never be able to be modified.

The algorithm shown in Figure 4.4 shows how this information can be extracted properly from the function which is being compiled. The detection of incoming parameters to the function can be done by checking for a use before a set (Line 13). Additionally, return values can be detected by looking at the outgoing live registers of all basic blocks that can exit the function (Line 25). Any parameters being passed to a called function are also unreplaceable and are found by inspecting all parameter line RTLs (Lines 6-7). Replacement information is further propagated to sibling nodes in the register interference graph (Line 32-33).

Other hardware registers that could not be replaced also needed to be detected and flagged. Since this de-optimization process was designed to be machine independent, additional machine-dependent information needed to be provided to indicate which registers should never be deallocated. For the ARM, such registers included the frame pointer, stack pointer, program counter, link register, and condition codes register. In the case of each register, they perform a function that is not able to be provided by any other register. Thus deallocating in these cases could never be beneficial.

4.3.2 Code Expansion and Global Variable Offsets

For the ARM architecture, loads of global variables can be handled by using a PC-relative immediate offset as the address. This offset is specified as a 12-bit value. An example instruction can be found in Line 1 of Figure 4.6. High level language compilers are free to place globals as needed and generate appropriate instructions for accessing globals that may be out of immediate reach. The generated assembly code from GCC attempts to place globals appropriately and switches to an indirect register mechanism for loading the address of a global if absolutely necessary. This poses a problem with direct assembly translation however, as any potential expansion in code size can push a symbolic offset just out of reach. Such a mistake will yield code that the assembler cannot handle.

During the translation process, there are certain block load and store instructions generated by GCC that must be expanded to multiple loads and stores in order to be handled correctly with VISTA. This expansion was enough to break two functions (*parse_switches* from *jpeg*, and *main* from *fft*) from all of the functions within the six tested programs for this study. The expansion problem is only exacerbated when performing de-optimizations on the assembly file, since each of the specified de-optimizations only modifies and adds RTLs while

never deleting any existing RTLs. If the resulting expanded code is not vastly improved by later optimization phases, it may contain global offsets that are out of reach and thus be unable to be assembled correctly.

Initial solutions to this problem included the construction and use of an additional peephole optimizer post-pass on the assembly files before assembling them. The peephole optimizations included were able to reconstruct a large number of the block memory access instructions, but not all. The *parse_switches* function was never able to be re-compressed properly even from optimized code, and thus this function was externally linked to the final executable and excluded from test results. Even with the peephole optimizer condensing instruction sequences, de-optimizations caused greater trouble, since many times the corresponding optimization phase was not reinvoked by VISTA using the genetic algorithm. This led to code that retained additional loads and stores or loop-invariant instructions, thus pushing global offsets out of bounds. It was eventually decided to eliminate this additional optimizer from the process. Instead, all phase sequences that produced code that could not be assembled were identified as invalid sequences and rejected by the genetic algorithm.

CHAPTER 5

REVERSE COPY PROPAGATION

This chapter discusses the optimization phase *reverse copy propagation* that has been added to the VISTA framework to enable additional benefits from the de-optimization of code. The first section describes the motivation for this optimization phase. Next we describe the implementation in detail. Finally we present an example showing *reverse copy propagation* in action.

5.1 Motivation for Reverse Copy Propagation

The re-performing of register assignment after de-optimization provides an opportunity for the compiler to improve on the original assembly code. An improved assignment of registers can reduce register pressure by freeing up valuable registers that will later enable further optimizations. However this opportunity works both ways and leads to the biggest potential problem, which is choosing a register assignment that inhibits further transformations that standard re-optimization can detect and perform.

The VPO compiler chooses register assignments based on what registers are available for the live range. If the live range does not cross a function call, then a scratch register is a good choice, since it need not be saved or restored. When a live range does cross a function call, non-scratch registers are best, since it will be the called function's responsibility to provide necessary save and restore code if such a register is used. The real problem exists when *register assignment* is performed with less information. After performing de-optimizations, *register assignment* works with extremely short register live range spans. These short spans usually then have scratch registers assigned to them. When performing additional optimization phases later, live ranges may be extended and the choice of a scratch register may be suboptimal since it will result in spills or register moves. The following

example demonstrates the shortcomings of re-performing *register allocation* before additional optimizations.

Figure 5.1 shows four sequences of RTLs at different stages of the optimization process, corresponding to the function *keymatch* from the *jpeg* benchmark. The topmost portion shows the initial input RTLs as translated from the GCC-generated assembly file (Lines a1-a3). There is a save of incoming parameter register **r[0]** (Line a1). This saved copy is then used to access data one byte at a time in a loop (Lines a2-a3). The second part of the figure shows the first few RTLs after performing de-optimizations as well as re-assigning registers (Lines b1-b6). It is noticeable that register **r[12]** is assigned the most work. It is constantly being loaded and stored, since it is the first available scratch register for the ARM and there are no intervening function calls. Additionally live range spans are relatively short for registers in the de-optimized state, typically being between 1 and 3 instructions. This is where the problem begins, since register **r[12]** is clearly not a good choice when these register live ranges are coalesced after re-performing *register allocation*.

The third section shows the RTLs after re-performing *register allocation* (Lines c1-c6). The local variable **_keymatch_0** has been allocated the register **r[8]**. All memory references to allocated variables are then replaced by the new registers (Lines c2, c3, c5, c6). In VPO, excess RTLs are cleaned up by performing additional optimization phases such as *instruction selection* and *dead assignment elimination*. In this case, the iteration of *instruction selection* causes several of the RTLs to be combined in the function. Lines c1-c3 can be combined and replaced with line d1. Lines c4 and c6 are combined to obtain Line d2. The bottom section in the figure shows the final results after re-performing *instruction selection* on the entire function. Clearly an additional RTL has been added that was not necessary in the original code (Line d3).

The standard version of VPO does not have the ability to perform a code-improving transformation that would allow the **r[12]** references in Lines d1-d3 to be changed to references to **r[8]**. Such a transformation can best be described as a *copy propagation* phase that works in reverse. The application of such a transformation in this particular case is obvious from just simple code inspection. Such a transformation would allow the removal of the RTL in Line d3 during *dead assignment elimination*, since it would result in a register to register copy using the same source and destination.

Translated Input RTLs			
Line	RTLs	Deads	Comments
a1	r[6]=r[0];	r[0]	Copy/save parameter r[0]
a2	r[4]=B[r[6]]&255;r[6]=r[6]+1;		Initial load of value
	...		r[6] live this entire time
a3	r[4]=B[r[6]]&255;r[6]=r[6]+1;		Load inside loop
	...		

De-optimized RTLs After Register Re-assignment			
Line	RTLs	Deads	Comments
b1	**r[12]**=r[0];	r[0]	Copy/save parameter r[0]
b2	R[r[13]+_keymatch_0]=**r[12]**;	**r[12]**	Interblock live range
b3	**r[12]**=R[r[13]+_keymatch_0];		so use loads/stores
b4	r[1]=B[**r[12]**]&255;**r[12]**=**r[12]**+1;		
b5	R[r[13]+_keymatch_0]=**r[12]**;	**r[12]**	
b6	R[r[13]+_keymatch_1]=r[1];	r[1]	
	...		

RTLs After Register Re-allocation			
Line	RTLs	Deads	Comments
c1	r[12]=r[0];	r[0]	Copy/save parameter r[0]
c2	**r[8]**=r[12];	r[12]	**r[8]** allocated for **_keymatch_0**
c3	r[12]=**r[8]**;	**r[8]**	
c4	r[1]=B[r[12]]&255;r[12]=r[12]+1;		
c5	**r[8]**=r[12];	r[12]	
c6	**r[6]**=r[1];	r[1]	**r[6]** allocated for **_keymatch_1**
	...		

RTLs After Instruction Selection			
Line	RTLs	Deads	Comments
d1	**r[12]**=r[0];	r[0]	Copy/save parameter r[0]
d2	r[6]=B[**r[12]**]&255;**r[12]**=**r[12]**+1;		Initial load of value
d3	r[8]=**r[12]**;	**r[12]**	Copy/save **r[12]**
	...		r[8] live this entire time
d4	r[4]=B[r[8]]&255;r[8]=r[8]+1;		Load inside loop
	...		

Figure 5.1: Register Re-assignment with Keymatch

5.2 Implementing Reverse Copy Propagation

Reverse copy propagation was designed to work independent of traditional *copy propagation* as implemented in VPO. *Copy propagation* is performed during the *common subexpression elimination* phase of VPO, so *reverse copy propagation* is added as a final pass during that phase as well. The algorithm for performing *reverse copy propagation* requires both forward and backward spanning searches into the function control flow graph. The complete process is defined by three algorithms. One is the main algorithm for actually performing *Reverse copy propagation*. The other two algorithms recursively assist in computing the fixed point of register live ranges and validating the transformations. They are *Backward Scan* and *Forward Scan* and are covered after describing the main algorithm. The primary algorithm behind *reverse copy propagation* is shown in Figure 5.2.

Reverse copy propagation works by examining each RTL in each basic block of a given function (Lines 1-2). When a register move is detected with a dead source register (the register value that is being copied), the algorithm will attempt to replace all references to that source register in prior RTLs with references to the destination register instead (Lines 3-25). To do this the algorithm keeps track of two lists: a backlist containing backward spanning references to the register and a forwardlist containing forward spanning references to the register. Initially we add the preceeding RTLs before the move instruction to the backlist for validity checking (Lines 6-7). If the move starts a basic block, then we must add all predecessor blocks to the backlist instead (Line 9).

At this point we have elements in the backlist, but no elements in the forwardlist. We now will add elements to each list in order to effectively capture the live range of the source register. The main loop of the optimization will terminate when either the replacement is seen as invalid or there are no further backward or forward spans to verify (Lines 12-18). Both **backscan** and **forwardscan** examine a given list element checking the RTLs for replacement validity (Lines 15,17). Both scanning functions will mark the *reverse copy propagation* invalid if there are register conflicts (Line 18).

If the entire live range of the register is able to be reconstructed without detecting any conflicts, the replacement can then proceed (Lines 19-25). Each element of the backlist and forwardlist are scanned for references to the source register, which are then replaced by the

```
1  foreach blk ∈ function basic blocks do
2    foreach rtl ∈ blk do
3      if rtl is a move with dead source register then
4        dst ← destination register of move
5        src ← source register of move
6        if rtl has preceeding rtls then
7          add preceeding rtl list to the backlist
8        else
9          add all predecessor blocks to the backlist
10       changes ← TRUE
11       valid ← TRUE
12       while changes and valid do
13         changes ← FALSE
14         foreach elem ∈ backlist do
15           changes | = backscan(elem)
16         foreach elem ∈ forwardlist do
17           changes | = forwardscan(elem)
18         valid ← FALSE if scanning detects conflicts
19       if valid then
20         foreach elem ∈ backlist do
21           foreach trtl ∈ elem→rtls do
22             replace all src references in trtl with dst
23         foreach elem ∈ forwardlist do
24           foreach trtl ∈ elem→rtls do
25             replace all src references in trtl with dst
```

Figure 5.2: Reverse Copy Propagation

destination register (Lines 22, 25). After all replacements are made, the algorithm proceeds to examine other potential candidates for *reverse copy propagation*.

Figure 5.3 shows the method in which backward scans are made. First the element is checked for previous backward scanning (Line 1). If it has not been seen before, it is marked as scanned backwards (Line 2). Now, each RTL in the block is examined in reverse, searching for potential conflicts (Lines 3-9). If an RTL sets or uses the destination register from the move instruction we are trying to eliminate, then there is a conflict, so the propagation is

invalid (Lines 4-6). If an RTL sets the source register from the move instruction but does not use that same source register, then all successor RTLs are added to the forwardlist and the function returns signifying a change to the scan lists (Lines 7-9). If neither terminating condition is reached, then the predecessors of the current basic block are examined (Lines 10-17) to determine validity. If the block has no predecessors, then clearly a conflict exists, as the move that we are attempting to eliminate contains an incoming parameter register (Lines 10-12). If there are predecessors however, the algorithm adds each predecessor to the backlist and recursively scans it (Lines 14-16). Any changes to the scanlists are propagated as a return value (Lines 16-17). If the element had already been scanned, no changes are returned (Line 18).

```
1  if elem has not been scanned backwards already then
2      mark elem as scanned backwards
3      foreach rtl moving backwards through elem→rtls do
4          if rtl sets or uses dst then
5              valid ← FALSE
6              return FALSE
7          else if rtl sets src and does not use src then
8              add successor rtls to the forwardlist
9              return TRUE
10     if elem has no predecessor blocks then
11         valid ← FALSE
12         return FALSE
13     changes ← FALSE
14     foreach pred ∈ elem→preds do
15         add pred to backlist
16         changes | = backscan(pred)
17     return changes
18 else return FALSE
```

Figure 5.3: Backward Scan

Forward scans are done in the manner shown in Figure 5.4. The element is checked for previous forward scans to prevent the main algorithm from looping forever (Line 1). Unmarked blocks are then marked as forward scanned (Line 2). Each RTL in the block is examined going forward to search for conflicts or the terminating condition (Lines 3-9). If the RTL shows the source register as being dead, the preceeding RTLs are added to the

backlist and the function returns showing that a change has been made to the scanning lists (Lines 4-6). If the RTL sets or uses the destination register, then the transformation is invalidated due to the conflict (Lines 7-9). If no terminating conditions are reached before the RTL list is exhausted, then we must examine the successors of the block (Lines 10-17). If the current block is an exit block then we are attempting to rename a return register which is not allowed (Lines 10-12). If this is not an exit block, then we will examine each successor block in turn (Line 14-16). Each successor block is added to the forward list and then scanned (Lines 15-16). Change information is collected and propagated back to the calling function (Lines 16-17). If the element had been previously scanned, no changes are returned (Line 18).

```
1  if elem has not been scanned forwards already then
2  |  mark elem as scanned forwards
3  |  foreach rtl moving forwards through elem→rtls do
4  |  |  if rtl has death of src then
5  |  |  |  add preceeding rtl list to the backlist
6  |  |  |  return TRUE
7  |  |  else if rtl sets or uses dst then
8  |  |  |  valid ← FALSE
9  |  |  |  return FALSE
10 |  if elem is exit block then
11 |  |  valid ← FALSE
12 |  |  return FALSE
13 |  changes ← FALSE
14 |  foreach succ ∈ elem→succs do
15 |  |  add succ to forwardlist
16 |  |  changes | = forwardscan(succ)
17 |  return changes
18 else return FALSE
```

Figure 5.4: Forward Scan

5.3 Reverse Copy Propagation Example

This section goes through a simple example of performing *reverse copy propagation* on a de-optimized and re-optimized function. For simplicity, the initial motivating example

of *keymatch* from *jpeg* is used. Figure 5.1 shows the original de-optimization and re-optimization causing code expansion. Figure 5.5 depicts several of the important steps in applying *reverse copy propagation* to the code. The top block (Lines a1-a4) show the starting code after being de-optimized and re-optimized. The first register move in not able to be transformed since it uses a register that is an incoming parameter (Line a1). There is another register move saving the scratch register **r[12]** for use in a later loop (Line a3).

To start the process of *reverse copy propagation*, we perform a backwards scan (Lines b1-b3). The preceeding RTLs (Line b2 backwards) are currently the only elements in the backlist. Line b2 shows use of the source register (**r[12]**) and line b1 shows the set of the source register. Now the following RTLs are added to the forwardlist (Line b2 forwards). The third box shows the process of forward scanning to determine any potential live range conflicts (Lines c1-c3). The scan begins on line c2 and continues through to line c3, where it sees the death of source register **r[12]**. At this point the transformation can proceed since we have determined that there is no conflict across the live range.

Lines d1-d3 show the replacement steps for *reverse copy propagation*. In line d1, the **r[12]** is replaced by the destination register **r[8]**. All other references to **r[12]** are also replaced by **r[8]** for the duration of the live range as determined by the forward and backward scanning. Line d3 shows the final necessary replacement, leaving a register move with the same source and destination. Notice that the **r[12]** reference in the dead register list has been removed. *Instruction selection* will later remove the unnecessary RTL in line d3, yielding the final code as shown in lines e1-e3.

Candidate for Reverse Copy Propagation			
Line	RTLs	Deads	Comments
a1	r[12]=r[0];	r[0]	Copy/save parameter r[0]
a2	r[6]=B[r[12]]&255;r[12]=r[12]+1;		Initial load of value
a3	**r[8]=r[12];**	**r[12]**	RCP candidate
	...		r[8] live this entire time
a4	r[4]=B[r[8]]&255;r[8]=r[8]+1;		Load inside loop
	...		

Backwards Scanning during Reverse Copy Propagation			
Line	RTLs	Deads	Comments
b1	**r[12]**=r[0];	r[0]	Scan sees the set of dst
b2	r[6]=B[**r[12]**]&255;**r[12]**=**r[12]**+1;		Scan sees use of src here
b3	r[8]=r[12];	r[12]	Start backwards scan
	...		Remainder of function

Forward Scanning during Reverse Copy Propagation			
Line	RTLs	Deads	Comments
c1	r[12]=r[0];	r[0]	Start forwards scan
c2	r[6]=B[**r[12]**]&255;**r[12]**=**r[12]**+1;		Scan sees use of src here
c3	r[8]=**r[12];**	**r[12]**	Scan sees death of src here
	...		Remainder of function

Replacement during Reverse Copy Propagation			
Line	RTLs	Deads	Comments
d1	**r[8]**=r[0];	r[0]	Replacement
d2	r[6]=B[**r[8]**]&255;**r[8]**=**r[8]**+1;		Replacement
d3	r[8]=**r[8];**		Replacement with elimination of death
	...		Remainder of function

After Further Optimizations			
Line	RTLs	Deads	Comments
e1	r[8]=r[0];	r[0]	Copy/save parameter r[0]
e2	r[6]=B[r[8]]&255;r[8]=r[8]+1;		Initial load of value
	...		r[8] live this entire time
e3	r[4]=B[r[8]]&255;r[8]=r[8]+1;		Load inside loop
	...		

Figure 5.5: Reverse Copy Propagation with Keymatch

CHAPTER 6

EXPERIMENTAL TESTING

This chapter presents the results of running the experiments described in Chapter 2. These experiments were designed to demonstrate the efficacy of performing de-optimizations before re-optimizing code. Actual statistics for running VISTA's genetic algorithm search for effective optimization sequences are compared both with and without de-optimizations in the first section. The second section presents an expanded discussion of the experimental results. This includes an analysis of some of the problems encountered that caused the implemented de-optimization phases to be less effective than originally anticipated.

6.1 Experimental Results

Each benchmark program was instrumented during compilation using the EASE framework to obtain both static code size and dynamic instruction execution counts. The GCC-generated code for each benchmark was translated to the RTL format and both static and dynamic counts are collected as baseline measures after performing a simple compilation pass with VISTA. This pass included performing *instruction selection* and *predication* during the *fix entry exit* phase. The GCC-generated code used as the baseline did have the potential to obtain a slight benefit in this case, since VISTA may detect additional sequences that are able to be predicated, as well as other instruction sequences that can be combined into fewer RTLs. The benchmark programs that were tested are described in Table 2.1.

Table 6.1 shows the results of running the experiments for the StrongARM architecture. Each of the six tested benchmarks (*bitcount*, *dijkstra*, *fft*, *jpeg*, *sha*, and *stringsearch*) is presented individually. The field labeled *Compiler Strategy* denotes whether just the genetic algorithm search was performed (`opt`) or if de-optimization phases were enabled prior to executing the genetic algorithm search (`de-opt`). Additionally, *reverse copy propagation* was added as an additional pass of the *common subexpression elimination* optimization phase.

Table 6.1. Effect of De-optimization on Static and Dynamic Instruction Count

Benchmark	Compiler Strategy	Opt. for Space		Opt. for Both		Opt. for Speed	
		static count	dynamic count	static count	dynamic count	static count	dynamic count
bitcount	opt	-2.32 %	0.00 %	-2.32 %	0.00 %	-2.32 %	0.00 %
	de-opt	-2.32 %	0.00 %	-2.32 %	0.00 %	-2.32 %	0.00 %
dijkstra	opt	-1.30 %	-2.70 %	-1.30 %	-2.70 %	-1.30 %	-2.70 %
	de-opt	-2.16 %	-2.73 %	-3.03 %	-2.73 %	-3.03 %	-2.73 %
fft	opt	-0.19 %	0.00 %	-0.19 %	0.00 %	-0.19 %	0.00 %
	de-opt	-0.19 %	-0.35 %	0.00 %	0.00 %	0.00 %	0.00 %
jpeg	opt	-4.30 %	-10.61 %	-4.30 %	-10.61 %	-4.30 %	-10.61 %
	de-opt	-5.20 %	-10.53 %	-4.97 %	-8.12 %	-4.94 %	-10.53 %
sha	opt	-5.99 %	-4.39 %	-5.99 %	-4.39 %	-3.89 %	-6.27 %
	de-opt	-5.69 %	-4.36 %	-5.69 %	-4.36 %	-2.69 %	-6.17 %
stringsearch	opt	-0.92 %	-0.09 %	-0.92 %	-0.09 %	-0.92 %	-0.09 %
	de-opt	-3.23 %	-0.09 %	-3.23 %	-0.09 %	-3.23 %	-0.09 %
average	opt	-2.50 %	-2.97 %	-2.50 %	-2.97 %	-2.15 %	-3.28 %
	de-opt	-3.08 %	-3.01 %	-3.16 %	-2.55 %	-2.65 %	-3.25 %

Measurements were taken using three different criteria for VISTA's genetic algorithm search for effective phase sequences. The three configurations that were tested varied the weight of potential tradeoffs such as code size and dynamic instruction count. The tested criteria for these experiments included optimizing for static code size (`Opt. for Space`), optimizing for dynamic instruction execution count (`Opt. for Speed`), and optimizing for a combination of the two, weighting each equally (`Opt. for Both`). The initial translated RTLs from the GCC-generated code are used as a baseline measure to which all tested configurations for de-optimization and genetic algorithm searching can be compared. Results are calculated by comparing experimental static and dynamic instruction counts to the initial static and dynamic instruction counts for the GCC-generated code for each benchmark. Differences are expressed in the table as percentages.

Results show that performing de-optimizations before re-optimizing allows for some potential benefits. In both the *dijkstra* and *bitcount* benchmarks, de-optimizing provides benefits for each of the three fitness criteria tested. De-optimizations also benefit the optimizing for space criteria for both the *fft* and *jpeg* benchmarks. In the cases of *bitcount* and

sha, de-optimizing proved to be detrimental, leading to situations where semantic information was unavailable to obtain further improvements to the code. Many of the functions in theses two benchmarks showed individual improvements due to de-optimization, however the negative impact was much higher from other functions.

Overall results show that by performing re-optimizations alone, VISTA was successful in decreasing static code size by an average of 2.50% and dynamic instructions counts by an average of 3.28% across all benchmarks when compared to the original GCC-generated code. De-optimizing before re-optimizing yielded even greater success with decreasing static code size, with an average of 3.16%, winning against re-optimization alone in 4 out of 6 cases. Decrease in dynamic instruction counts reached an average of 3.25%, thus falling slightly short of the benefit of pure re-optimization. In all cases, re-optimization with or without de-optimization yields improvements to the original GCC-generated code. A closer look at the data shows that no primary or secondary fitness measure performs worse than the baseline GCC-generated code for an entire benchmark although some functions did experience an increase in code size when compiling for speed or an increase in executed instructions when compiling for reduced space. Such increases were later canceled out by other functions in the benchmark that benefited from decreased space requirements.

6.2 Discussion of Results

Looking at the results of these experiments, there are several critical points to note. De-optimization before re-optimization benefited the code in several cases, while detracting from performance in other cases. Despite the fact that not every benchmark saw improvement, the experiments produced some very interesting results. This section will provide an overview of the knowledge gained from running the de-optimization experiments. Several factors contributed to the performance of the de-optimization strategy including various interactions with other optimization phases, inherent randomness with the genetic algorithm, and the lack of more detailed dynamic execution statistics. Each of these items will be discussed in this section, as well as potential methods for improving the de-optimization strategy to overcome these problems.

6.2.1 Re-optimization Benefits

The GCC-generated code clearly benefited from the performing of additional optimization phases. One benefit of assembly translation and re-optimization is the potential to apply different optimizations that were not available or applied to the original code. Sometimes it is just a different implementation and set of heuristics for an optimization that can improve existing optimized code. As an example, VISTA is able to predicate several additional RTLs, leading to decreased code size (due to the lack of extra branch instructions) and potentially decreased dynamic execution counts (for predicates that are frequently true).

6.2.2 Effectiveness of Register Re-assignment

Examining the results of several benchmark runs, it became clear that register re-assignment played a critical role in the performance of the de-optimization strategy. Initial results without *reverse copy propagation* showed that de-optimization could not compete with pure re-optimization of code. The addition of *reverse copy propagation* helped in eliminating problems due to register re-assignment while also providing the pure re-optimization strategy with benefits as well. Despite improving the pure re-optimization strategy, de-optimization benefited enormously, overcoming just re-optimization in many cases for static code size and dynamic instruction count.

6.2.3 Genetic Algorithm Behavior

One area of confusion found in Table 6.1 is the comparison of static code size for the *jpeg* benchmark when looking at code that is de-optimized. For the mixed static/dynamic search criteria, the improvement in static code size (-4.97%) is not as good as optimizing for code size alone (-5.20%) considering that the code size only results also yield better dynamic counts. There are other portions of the table that exhibit similarly confusing results.

For these benchmarks, it would appear that the search criteria was ignored to some extent, or that there is a fundamental flaw in the VISTA genetic algorithm search for effective optimization phase sequences. However, this effect can be explained due to the inherent randomness of evolutionary algorithms. Random choices are made for initial population chromosomes, crossover pairs and mutations during the compilation of every function. The

application of a genetic algorithm is an attempt to trim a large search space in a logical manner, consistent with the fitness criteria. In each peculiar case shown in the results, the better sequences for a particular fitness criteria were just never uncovered in the search process.

There are several ways to limit the effects of randomness on the resulting compiled functions. One method is to adjust the variable configuration options for the genetic search algorithm. The sequence length can be made longer, to create more opportunities for re-optimizing code. The population size and the number of generations can be increased to allow for an even greater number of different chromosomes (phase orderings) to be tested. However the process still remains dependent on randomness, hence the best solution for one fitness criteria may not be found during a search using that particular fitness criteria.

6.2.4 Dynamic Execution Statistics

One of the biggest potential benefits of de-optimizing *register allocation* is the ability to change which variable live ranges will reside in memory and which will reside in registers. Register movement operations can be faster than memory movement operations like loads and stores. However, the dynamic instruction count obtained by VISTA only measures the total number of instructions executed, and thus loads and register move operations count the same. In general, *register allocation* replaces loads and stores with register move operations. It is very common for register move operations to become redundant after being manipulated via *common subexpression elimination* and other transformations. VISTA relies on code improving transformations such as *instruction selection* and *dead assignment elimination* to clean up the code by combining and removing any unnecessary RTLs.

Several functions had local variable allocations that differed from the original allocation performed by GCC. Since de-optimizations can sometimes suffer from the register re-assignment problem, register moves to save scratch registers (such as `r[12]`) can remain in the code. When executing this code for dynamic measures, what was once a store may now show up as a move. The dynamic count will remain the same, even though inspection could show this as a winning situation. Dynamic instructions counts also do not take into account any effects from the memory hierarchy. Cache hits and misses can greatly impact dynamic performance. Some instructions may even require multiple cycles or have

misprediction penalties associated with branching. VISTA supports optimizations such as *strength reduction* that can help to alleviate the increased cost of powerful instructions by replacing them with a sequence of cheaper instructions. Such a sequence might also go unnoticed as an improvement over the original complex instruction.

The best way to correct these problems is to obtain more accurate information concerning the dynamic execution behavior of the function. One particular method for obtaining precise behavior information is through the use of hardware simulation. By simulating all characteristics of the target machine (StrongARM), statistics including cycle-accurate execution times may be extracted. Additionally, behavioral characteristics of the memory hierarchy may be obtained, such as the actual cache hit rate, or the page fault count. VISTA can then be modified to interface with such a simulator to obtain more accurate fitness testing when searching for the most effective optimization phase sequence.

CHAPTER 7

RELATED WORK

This chapter focuses on research areas that overlap in some manner with this work. To the best of our knowledge, there has been no other research in using de-optimizations to enhance the re-optimization of assembly code. However there has been related work in the fields of assembly translation and de-optimization. Different aspects of these areas are used to facilitate reverse engineering, legacy binary translation, link-time optimizations, as well as the development of symbolic debuggers that can handle optimized code.

Binary translation is the process of converting an executable program from one particular platform to a different platform. Platform differences can include instruction set, application binary interface (ABI) specifications, operating system, and executable file format. The translation of binary programs has particular importance in the porting of legacy software for which source code is unavailable to new hardware. There are a few key differences between binary translation and the ASM2RTL tools discussed in this work. With binary translators, the stack layout remains constant and thus arrays and structures cannot be accidentally placed incorrectly. This also limits the effectiveness of re-optimizations since even scalar local variables cannot be adjusted. Additionally, the ASM2RTL tools work at the level of assembly code and not binary executables.

Existing systems that perform binary translation include the Executable Editing Library (EEL) and the University of Queensland Binary Translator (UQBT). EEL is a library developed at the University of Wisconsin to facilitate the construction of tools that edit executables such as binary translators and optimizers [16]. EEL has also been used to construct programs that instrument executables for the purpose of gathering performance data. UQBT is a binary translator developed at the University of Queensland that was designed to be easily retargeted [5]. Similar to this work, UQBT accepts SPARC and ARM input files among other formats. UQBT features several backends generating various types of

output suitable for translation purposes, including Java virtual machine language (JVML), C, object code and VPO RTL. Although UQBT did have some success with smaller programs using the RTL backend, the overall RTL target required too much analysis information to be useful within the UQBT framework. Having additional analysis information being supplied statically to ASM2RTL before the start of the translation process helps to alleviate some of the problems encountered by UQBT. Additionally, ASM2RTL can always fall back on translations that maintain safety at reduced opportunity for improvement such as keeping all local variables in the same initial order on the stack.

Link-time optimizations are closely related to the process of binary translation, in that they both operate on a similar amount of semantic content. Link-time optimizations focus on interprocedural opportunities that were not able to be addressed during the compilation of individual functions or modules. Alto is a link-time optimizer targeting the Compaq Alpha that features optimizations such as *register allocation*, *inlining*, *instruction scheduling*, and *profile-directed code layout* [17]. In contrast, the de-optimization and re-optimization components of VISTA are designed to work with the function as a basic unit. Link-time optimizations could later be applied to the resulting re-optimized functions to further improve performance.

The process of de-optimizing code has been applied for several purposes, ranging from the debugging of optimized code to the reverse engineering of executable files. Symbolic debuggers are one of critically valuable tools of a programmer. Due to the use of optimizing compilers and the development of more complicated optimizations, debuggers have a need for additional information to provide proper interactive feedback when working with optimized executables. Hennessy addressed the problem of noncurrent variables when symbolically debugging optimized executables [10]. These are variables that may not contain their correct *current* value when walking through the optimized executable, perhaps due to *common subexpression elimination* or *dead assignment elimination*. Hölzle, Chambers and Ungar presented another approach for debugging optimized code using dynamic de-optimization [11]. Their system however was limited to only debug the SELF object-oriented language and also requires the original compiler to instrument the executable with all necessary de-optimization information at various *interrupt points* within the program.

Reverse engineering is the technique of extracting high-level source information from binary executable files. The reconstruction of control flow graphs (CFGs) is a primary step in the reverse engineering process, allowing input assembly instructions to be better represented as the common loop and test constructs seen in a high-level language. CFGs for binary code also tend to be more complicated than corresponding CFGs for the initial high-level source code. This increased complexity is only further exacerbated by optimizations that modify control flow such as *predication*, *speculation* and *instruction scheduling*. Snavely, Debray and Andrews have experimented with performing de-optimizations on CFGs generated by reverse engineering Itanium executables [19]. Results indicate that de-optimizations allow for reductions in both complexity and size of the generated CFGs.

CHAPTER 8

ADDITIONAL BENEFITS OF THIS WORK

There are several other projects that are currently using portions of this work for different purposes. This chapter discusses two particular instances. One is the use of the TI TMS320c54x-based translator from ASM2RTL in DSP-driven laboratory experiments for students. Another use of this work is found in the current implementation of VISTA's genetic algorithm search for effective optimization sequences. In particular, register interference graphs are constructed and used to detect similarities between generated code for different phase orderings.

8.1 Use of ASM2RTL in DSP-driven Labs

Currently the course **ECE 320: Digital Signal Processing Laboratory** employs the use of the ASM2RTL tool suite and a modified VISTA compiler [12]. The course is taught by Dr. Douglas Jones at the University of Illinois at Urbana-Champaign. It is an undergraduate course focusing on concepts for DSP (Digital Signal Processor) systems and software development. Laboratory experiments cover topics such as FIR (Finite Impulse Response) and IIR (Infinite Impulse Response) filtering, sampling, multi-rate processing, elliptical low-pass filters, FFTs (Fast Fourier Transform), and quadrature phase-shift keying.

Students write code for each laboratory experiment in C or in assembly. For the assembly code, students can then use the ASM2RTL component known as ti2rtl to convert the source into VISTA RTL format for further optimization. At this point, the VISTA compiler framework can be used to fine tune and further optimize the code. Students can step through the application of various optimizations using the VISTA viewer, gaining some insight into the process of improving assembly code. Additionally, the students can specify hand transformations to the code that could not be automatically detected by VISTA.

8.2 Detecting Homomorphic Code in VISTA

Traditionally genetic algorithms have used hash tables to keep track of already computed solutions, in order to eliminate any unnecessary and costly steps involved in testing the fitness of a particular chromosome. The portion of the genetic algorithm search process in VISTA that takes the longest time is the actual assembling, linking and running of the compiled code. The VISTA system employs several hash tables to eliminate redundant executions of the time-consuming portions of the genetic algorithm [14].

VISTA employs four methods to detect previously tested functions. Each method detects a superset of the preceding method's matches, but requires additional analysis and checking time. Thus each hash table check is performed individually and if a match is ever found, the performance results stored in the table are used for the current sequence. VISTA first checks to see if the current optimization phase sequence has already been tested. If not, then the phases are applied. Phases that alter the code representation are classified as active. Active phases in the current sequence are then searched for in a different hash table. If this set of active sequences has not been tested, then the next step is to examine the function RTLs. The RTL sequence is checked for exact matches with previously tested functions via the use of a cyclic redundancy code (CRC) checksum.

The final redundancy check performed by VISTA is for equivalent or homomorphic code. This can be described as code that is structurally similar, but with a differing assignment of register names. This check employs a register interference graph (RIG) as described in Chapter 4. After the RIG is constructed, hardware register live ranges are all re-mapped to pseudo-registers as the basic blocks of the function are traversed in order. After performing the re-mapping, this new pseudo-register variant of the RTLs has its CRC computed. Any other matching CRC from the corresponding hash table will show the current optimization phase sequence as producing code that is equivalent to another sequence.

Figure 8.1 depicts register re-mapping for the purpose of homomorphic code detection. The top portion shows *register allocation* before *loop-invariant code motion*, and the bottom portion shows the reverse. Comparing the original RTLs for each example shows that the live ranges of **`r[1]`**, **`r[2]`**, and **`r[3]`** are different. These registers are displayed in bold in the example. If each register live range however is replaced with the next numeric

Register Allocation Before Code Motion			
Line	Label	RTLs	Re-mapped RTLs
a1		**r[2]**=0;	**r[32]**=0;
a2		r[0]=L21;	r[33]=L21;
a3		**r[3]**=4000+r[0];	**r[34]**=4000+r[33];
a4	L3		
a5		**r[1]**=R[r[0]];	**r[35]**=R[r[33]];
a6		**r[2]**=**r[2]**+**r[1]**;	**r[32]**=**r[32]**+**r[35]**;
a7		r[0]=r[0]+4;	r[33]=r[33]+4;
a8		c[0]=r[0]?**r[3]**;	c[0]=r[33]?**r[34]**;
a9		PC=c[0]<0,L3;	PC=c[0]<0,L3;

Code Motion Before Register Allocation			
Line	Label	RTLs	Re-mapped RTLs
b1		**r[3]**=0;	**r[32]**=0;
b2		r[0]=L21;	r[33]=L21;
b3		**r[1]**=4000+r[0];	**r[34]**=4000+r[33];
b4	L3		
b5		**r[2]**=R[r[0]];	**r[35]**=R[r[33]];
b6		**r[3]**=**r[3]**+**r[2]**;	**r[32]**=**r[32]**+**r[35]**;
b7		r[0]=r[0]+4;	r[33]=r[33]+4;
b8		c[0]=r[0]?**r[1]**;	c[0]=r[33]?**r[34]**;
b9		PC=c[0]<0,L3;	PC=c[0]<0,L3;

Figure 8.1: Two Functions with Homomorphic Code

pseudo-register, then the mapping shown on the right is obtained for each function. It is easy to see from this mapping that both functions are in fact structurally equivalent besides the different assignment of registers.

CHAPTER 9

FUTURE WORK

This chapter discusses potential areas for further investigation into the effectiveness of de-optimization on the re-optimization process. Improvements can be made in the testing of the de-optimization process or to the de-optimization process itself. Several improvements to the experiments can be made, including experimenting with a greater number and variety of benchmarks, as well as working with actual hand-tuned assembly code. The de-optimization process itself could be improved by the addition of further de-optimizations, mechanisms for selecting the amount and types of de-optimization, as well as specialized code-improving transformations designed to enhance de-optimized code.

9.1 More Extensive Experiments

In this study, six benchmarks from the MiBench embedded benchmark suite were selected for comparison. Results indicated that improvements are possible, but hard to come by. Working with additional benchmarks could potentially show the benefits of the de-optimization process more clearly. A large number of embedded applications are both programmed and optimized purely in assembly code. The use of industry-grade hand-written assembly code could lend additional credibility to the effectiveness of de-optimization before re-optimizing code. De-optimizations may even be more effective on hand-written code, where there is a possibility that suboptimal choices involving optimization can be very detrimental to the actual code. For example, poor register assignment for temporary values can lead to increased register pressure, which inhibits many code-improving transformations. Additionally, it has already been noted that even though hand-tuning of assembly code can provide improvements that a compiler cannot, such methods require a great deal of programmer time and thus cannot be expected to test a large number of different potential transformation orderings. De-optimization and re-optimization could be used

to improve hand-written assembly code without requiring an extraordinary amount of additional programming time.

Chapter 6 discusses some of the problems encountered when using dynamic instruction count to measure the executable performance of compiled code. VISTA can be extended to interface with a simulator for the StrongARM, thus providing greater accuracy in determining the fitness value for a particular phase ordering. The SimpleScalar tool-set is a popular and widely-used simulation tool for gathering accurate performance data [3]. Recently it has been extended to support the StrongARM processor, providing accurate cycle timing as well as detailed statistics about memory hierarchy performance. Such information could help to better distinguish the effect of performing each optimization during the genetic algorithm search of VISTA.

9.2 Improving De-optimization

This work focuses on two de-optimizations, specifically targeting the actual optimization phases *loop-invariant code motion* and *register allocation*. These two optimizations were chosen for their complementary nature as well as the observation that each performs a role that requires and consumes registers in the resulting compiled code. Thus the order in which these two optimizations are performed during the compilation of a function can produce dramatically different results. There are other optimization phases that also have a different impact depending on the phase order, and thus may benefit from de-optimization before being re-optimized. *Common subexpression elimination* is one particular phase that would be a good candidate for potential de-optimization. Additionally, the selection of which de-optimizations to enable for a particular function could be made into an adjustable parameter within VISTA. Thus de-optimizations which do not enable any further benefits for a function will be selected against by the genetic algorithm. Additionally, such a selection could turn off de-optimizations entirely if they prove to be disadvantageous, as they were for several of the functions tested in this study.

Another potential enhancement is the creation of further specific code-improving transformations to enable de-optimizations to regain potential lost benefits. The de-optimization of *register allocation* and *register assignment* showed poor performance when scratch registers

were used during the re-assignment of registers. A *reverse copy propagation* phase was developed to correct for this particular situation. Additional phases could be designed and implemented to enable poorly de-optimized code to attain its former representation at worst, and might further expose additional opportunities for improvement.

CHAPTER 10

CONCLUSIONS

Embedded systems development is dominated by requirements which can include rigid constraints on code size, power consumption, and time. It is common then for applications to be written in assembly code and hand-tuned to meet these expectations. Yet, just as traditional optimizing compilers are subject to the phase ordering problem, hand-generated assembly code can experience an analogous problem depending on programmer choices during the tuning process. The undoing of prior optimizations, or de-optimization of assembly code is one potential method of alleviating the negative effects of the phase ordering problem in such cases. This thesis has presented an extension of the VISTA framework suitable for investigating the effectiveness of de-optimizations on the re-optimization of previously generated assembly code.

There were several interesting results concerning the implementation and use of de-optimizations prior to re-optimizing assembly code. De-optimizations that reduce register pressure, such as *loop-invariant code motion* and *register allocation* were selected for evaluation, since registers are typically a limited resource in embedded systems. Results from the experiments performed showed that de-optimizations could be very beneficial in the re-optimization process, although it can potentially be detrimental as well. The added optimization phase *reverse copy propagation* allowed de-optimization to reattain previous code quality in particular sequences where register re-assignment produced code that could not be further optimized. Overall, it has been shown that de-optimizing assembly code can provide an additional opportunity for reordering optimization phases that may have already been performed on previously generated assembly code.

We have also shown that assembly translation is both a viable and a suitable method for the interconnection of different optimizing compilers in an iterative fashion. The ASM2RTL translator suite was designed and constructed for converting assembly code to VISTA RTL

format. ASM2RTL contains support for several assembly languages, but the StrongARM was chosen for experimentation. De-optimization and re-optimization of GCC-generated code using the genetic algorithm search features of VISTA provided decreases in static code size on average of 3.16% and decreases in dynamic instruction count on average of 3.25% when compiling for each individually. Additional code optimizers and corresponding assembly translators can be constructed for specific environments and ordered in a manner that provides the greatest benefit to the resulting code. In the embedded devices arena, more complex and longer optimization processes are acceptable since a large number of units are typically produced and code requirements may be more stringent than with traditional applications. The development of tools such as the ASM2RTL translator suite opens up new possibilities for the further optimization of code, particularly for hand-tuned assembly and legacy applications.

REFERENCES

[1] Benitez, M. E., and Davidson, J. W. A portable global optimizer and linker. In *Proceedings of the SIGPLAN'88 conference on Programming Language Design and Implementation* (1988), ACM Press, pp. 329–338.

[2] Benitez, M. E., and Davidson, J. W. Target-specific global code improvement: Principles and applications. Tech. Rep. CS-94-42, 4, 1994.

[3] Burger, D., and Austin, T. M. The simplescalar tool set, version 2.0. Tech. Rep. 1342, University of Wisconsin - Madison, Computer Science Dept., June 1997.

[4] Chow, F. C., and Hennessey, J. L. Register allocation by priority-based coloring. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction* (June 1984), pp. 222–232.

[5] Cifuentes, C., Van Emmerik, M., Lewis, B. T., and Ramsey, N. Experience in the design, implementation and use of a retargetable static binary translation framework. Tech. Rep. TR-2002-105, Sun Microsystems Laboratories, January 2002.

[6] Cooper, K. D., Schielke, P. J., and Subramanian, D. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems* (1999), ACM Press, pp. 1–9.

[7] Davidson, J. W., and Whalley, D. B. A design environment for addressing architecture and compiler interactions. *Microprocessors and Microsystems 15*, 9 (November 1991), 459–472.

[8] Free Software Foundation. Gnu compiler collection 3.3. http://gcc.gnu.org/, 2004.

[9] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. Mibench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization* (December 2001).

[10] Hennessy, J. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 323–344.

[11] Hölzle, U., Chambers, C., and Ungar, D. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 1992).

[12] Jones, D., Kramer, M., Berry, M., Wade, B., Moussa, D., Haun, M., Appadwedula, S., Janevitz, J., and Sachs, D. ECE 320 Digital Signal Processing lab notes. http://cnx.rice.edu/content/col10078/latest/, 2004.

[13] Kulkarni, P. Performance driven optimization tuning in vista. Master's thesis, Florida State University, Tallahassee, Florida, 2003.

[14] Kulkarni, P., Hines, S., Hiser, J., Whalley, D., Davidson, J., and Jones, D. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (2004), pp. 171–182.

[15] Kulkarni, P., Zhao, W., Moon, H., Cho, K., Whalley, D., Davidson, J., Bailey, M., Paek, Y., and Gallivan, K. Finding effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems* (2003), pp. 12–23.

[16] Larus, J., and Schnarr, E. Eel: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (1995).

[17] Muth, R., Debray, S., Watterson, S., and De Bosschere, K. alto: A link-time optimizer for the compaq alpha. *Software - Practice and Experience 31* (January 2001), 67–101.

[18] Rohou, E., Bodin, F., Seznec, A., Le Fol, G., Charot, F., and Raimbault, F. SALTO : System for assembly-language transformation and optimization. In *Proceedings of the 6th Workshop on Compilers for Parallel Computers* (December 1996), no. RR-2980, pp. 261–272.

[19] Snavely, N., Debray, S., and Andrews, G. Unscheduling, unpredication, unspeculation: Reverse engineering itanium executables. In *Proceedings of the 2003 Working Conference on Reverse Engineering* (November 2003), pp. 4–13.

[20] Vegdahl, S. R. Phase coupling and constant generation in an optimizing microcode compiler. In *Proceedings of the fifteenth annual workshop on microprogramming* (1982), pp. 125–133.

[21] Whitfield, D. L., and Soffa, M. L. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS) 19*, 6 (1997), 1053–1084.

[22] Zhao, W., Cai, B., Whalley, D., Bailey, M. W., van Engelen, R., Yuan, X., Hiser, J. D., Davidson, J. W., Gallivan, K., and Jones, D. L. Vista: a system for interactive code improvement. In *Proceedings of the joint conference on Languages, Compilers, and Tools for Embedded Systems* (2002), ACM Press, pp. 155–164.

BIOGRAPHICAL SKETCH

Stephen R. Hines

Stephen Hines was born on March 28, 1979 in Staten Island, New York. He attended the Illinois Institute of Technology, graduating with high honors in the Spring of 2001 with a Bachelor of Science Degree in Computer Engineering. In the Summer of 2004, he graduated from The Florida State University with a Master of Science Degree in Computer Science. Currently he is pursuing his Ph.D. in Computer Science at The Florida State University under the direction of Dr. David Whalley. His main areas of interest are compilers, computer architecture, and embedded systems.