Understanding Optimization Phase Interactions to Reduce the Phase Order Search Space

Michael R. Jantz

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas School of Engineering in partial fulfillment of the requirements for the degree of Master of Science.

Thesis Committee:

Dr. Prasad Kulkarni: Chairperson

Dr. Perry Alexander

Dr. Andy Gill

Date Defended

The Thesis Committee for Michael R. Jantz certifies
That this is the approved version of the following thesis:

Understanding Optimization Phase Interactions to Reduce the Phase Order Search Space

Committee:

_____

Chairperson

_____

_____

_____

Date Approved

Acknowledgements

I would like to thank my advisor, Professor Prasad Kulkarni, for mentoring me during my time on this project. He often goes above and beyond his responsibilities as a professor and without his guidance none of this would have been possible.

I would like to thank all the other professors who have taught and advised me during my time at the University of Kansas. I would especially like to thank Professor Nancy Kinnersley, who advised me on my way to graduate school, and Professors Perry Alexander and Andy Gill, who, along with my labmates, have made me feel welcome in the Computer Systems Design Laboratory.

I would like to thank the ITTC help team (especially Chuck Henry) for their excellent system administration and for introducing me to the cluster here at ITTC. Without this computing power, I would not have been able to complete many of my experiments.

Finally, I would like to thank my parents and sisters for all of their love and support. This means the world to me.

Thank you all.

Abstract

Compiler optimization phase ordering is a longstanding problem, and is of particular relevance to the performance-oriented and cost-constrained domain of embedded systems applications. Optimization phases are known to *interact* with each other, enabling and disabling opportunities for successive phases. Therefore, varying the order of applying these phases often generates distinct output codes, with different speed, code-size and power consumption characteristics. Most current approaches to address this issue focus on developing innovative methods to selectively evaluate the vast phase order search space to produce a good (but, potentially suboptimal) representation for each program.

In contrast, the goal of this thesis is to study and reduce the phase order search space by: (1) identifying common causes of optimization phase interactions across all phases, and then devising techniques to eliminate them, and (2) exploiting natural phase independence to prune the phase order search space. We observe that several phase interactions are caused by *false* register dependence during many optimization phases. We explore the potential of cleanup phases, such as *register remapping* and *copy propagation*, at reducing false dependences. We show that innovative implementation and application of these phases not only reduces the size of the phase order search space substantially, but can also improve the quality of code generated by optimizing compilers. We examine the effect of removing cleanup phases, such as *dead assignment elimination*, which should not interact with other compiler phases, from the phase order search space. Finally, we show that reorganization of the phase order search into a multi-staged approach employing sets of mutually independent optimizations can reduce the search space to a fraction of its original size without sacrificing performance.

Contents

Abstract **iii**

Table of Contents **iv**

List of Figures **vi**

List of Tables **vii**

1 Introduction **1**

2 Related Work **6**

3 Background **10**

3.1 *Attempted* Optimization Phase Order Space 10

3.2 Categorization of the Phase Ordering Problem 11

3.3 Our Approach 12

4 Experimental Setup **15**

4.1 The VPO Compiler 15

4.2 Optimization Space Details 16

4.3 Our Benchmark Set 20

4.4 Setup for Search Space Exploration 20

4.5 The Testing Framework 22

5 False Phase Interactions **24**

5.1 Examples of Phase Interactions 25

5.2 Effect of Register Pressure on Phase Order Space and Performance 27

5.3 Measuring the Effect of False Register Dependence 30

5.3.1 Reg. Remapping to Reduce the Phase Order Search Space 30
5.3.2 Copy Propagation to Reduce the Phase Order Search Space 34
5.3.3 Combining Register Remapping and Copy Propagation . . 36
5.4 Eliminating False Register Dependence on Real Architectures . . 37
5.4.1 Reducing the Search Space with Copy Propagation 37
5.4.2 Improving Performance with Localized Register Remapping 39

6 Phase Independence 44
6.1 Eliminating Cleanup Phases 45
6.2 Eliminating Branch Optimizations 48
6.3 Multi-stage Phase Order Searches 51
6.3.1 The Multi-Stage Approach 52
6.3.2 Automatic Set Partitioning 53

7 Future Work 57

8 Conclusions 59

References 61

List of Figures

3.1 Optimization phase order search space DAG 13

5.1 True phase interaction example 25
5.2 *Register remapping* eliminates false register dependence 26
5.3 *Copy propagation* eliminates false register dependence 26
5.4 Search space sizes with different numbers of available registers . . 29
5.5 Effect of *register remapping* (512 registers) 31
5.6 Non-orthogonal use of registers affects register-remapped code . . 33
5.7 Effect of *copy propagation* (512 registers) 35
5.8 Effect of *register remapping* with *copy propagation* (512 registers) 36
5.9 Effect of *copy propagation* (16 registers) 38
5.10 Effect of *aggressive localized register remapping* (16 registers) . . . 40
5.11 Effect of *conservative localized register remapping* (16 registers) . . 41
5.12 Effect of *localized register remapping* on batch performance 42
5.13 *Register remapping* disables some optimization opportunities . . . 43

6.1 Effect of removing *DAE* 45
6.2 Non-orthogonal use of registers affects space with *DAE* removed . 46
6.3 Effect of removing branch optimizations 49
6.4 Branch optimization interaction 50
6.5 Effect of multi-stage phase order search 52
6.6 Independence table 54

7.1 Phase interaction between *instruction selection* and *CSE* 58

List of Tables

3.1 Example phase order search space 11

4.1 Candidate Optimization Phases Along with their Designations . . 17
4.2 MiBench Benchmarks Used in the Experiments 21

Chapter 1

Introduction

Compiler optimization phase ordering and selection have been longstanding and persistent problems for compiler writers and users alike [9, 23, 26]. Current compilers typically contain several different optimization phases. Each optimization phase applies a sequence of transformations to improve the quality of the generated code for some measure of performance, such as speed, code-size or power consumption. Optimization phases require specific code patterns and/or availability of architectural registers to do their work. Consequently, phases interact with each other by creating or destroying the conditions necessary for the successful application of successive phases. Unfortunately, no single ordering of optimization phases is able to produce the best code for all programs in any investigated compiler [7, 16, 20, 25, 27]. Instead, the ideal phase sequence depends on the characteristics of the code being compiled, the compiler implementation, and the target architecture.

Most conventional compilers are plagued with the problem of determining the ideal sequence or ordering of optimization phases to apply to each function or program so as to maximize the gain in either speed, code-size, power, or any

combination of these performance constraints. Furthermore, the potentially large performance difference between the code produced by different phase sequences can have major implications on the cost (e.g., memory size) or power requirements of the application. Such implications make finding a good set and ordering of optimization phases very attractive in performance-critical application domains such as embedded systems.

The most common solution to the phase ordering problem employs iterative search algorithms to evaluate the performance of the codes produced by many different phase sequences and select the best one. Although this process may take longer than traditional compilation, longer compilation times are often acceptable when generating code for embedded systems. Many embedded system developers attempt to build systems with just *enough* compute power and memory as is necessary for the particular task. Most embedded systems are also constrained for power. Thus, reducing the speed, code size, and/or power requirements is extremely crucial for embedded applications, as reducing the processor or memory cost can result in huge savings for products with millions of units shipped.

However, the large number of optimization phases typically available in current compilers results in extremely large phase order search spaces that are either infeasible or impractical to exhaustively explore [22]. Optimization phase ordering/selection search spaces in current compilers have been reported to consist in excess of 15^{32} [18], or 16^{10} [2], or 2^{60} [14] unique phase sequences. Therefore, reducing the compilation time of iterative phase order space search is critical to harnessing the most benefits from modern optimizing compilers. Shorter iterative compilation times can be accomplished by two complementary approaches: (1) develop techniques to reduce the phase order search space itself, or (2) devise new

search strategies that can perform a more *intelligent* but partial exploration of the search space. Most recent and existing research effort is *solely* focused on the second approach, and attempts to employ statistical or machine learning methods, along with enhanced correlation techniques to restrict the number of phase sequences that are reached and require evaluation during the iterative search. By contrast, the goal of this research is to analyze and address the most common optimization phase interactions, and then to develop solutions that can substantially reduce the phase order search space. We believe that such reduction will not only make exhaustive phase order searches more practical, but also enable better predictability and efficiency for intelligent heuristic searches.

Registers are an important resource during the application of many optimization phases, especially in a compiler backend. It is well recognized that the limited number of registers causes several phase interactions and ordering issues [3, 13]. Interestingly, our analysis of the phase order search space found that many such phase interactions are not caused by register contention, but exist due to *false* dependences between and during phase transformations that reuse the same register numbers. In this study, we devise approaches to explore the extent and impact of this issue on the phase order search space size and generated code performance. We find that techniques to reduce false register dependences *between* phases has a huge limiting effect of the size of the search space. Furthermore, we find that reducing false dependences *during* phases can also provide additional optimization opportunities and result in improving the quality of the code produced by such phases. Thus, the work presented in this thesis shows promise to not only improve the state of iterative compilation but also provide *guidelines* for compiler implementations to generate higher-quality code.

Another issue is how to organize exhaustive phase order searches to exploit independent phases to drastically reduce the size of the search space while still generating code that performs as well as the code produced by the naïve phase order search. We investigate two complementary techniques to reorganize the phase order searches: eliminate cleanup phases from the phase order search, and multi-stage phase order searches with mutually independent sets of phases in each stage. Cleanup phases, such as *dead assignment elimination* and *dead code elimination* (see Table 4.1), assist other phases by cleaning junk instructions and blocks left behind by other code transforming optimizations. As such, we attempted to remove such phases from the set used during the naïve phase order searches and implicitly apply them after every phase. We find that applying cleanup phases outside the phase order search does not affect the quality of the code generated in most cases, but results in substantially smaller exhaustive phase order search spaces.

For our next technique, we first employ compiler writers' knowledge to partition phases into two sets: branch and non-branch optimization phases. Generally, branch optimizations may interact with other branch phases, but do not interact with other non-branch optimization phases. We developed a multi-stage exhaustive phase order search algorithm that applies only the branch phases in the first stage and the branch and non-branch phases in the second stage. We find that partitioning the phase order search into searches with branch and non-branch optimizations can reduce the phase order search to a fraction of its original size while still achieving the best overall performing function instance. We then explore methods for automatically generating set partitions for a multiple stage phase order search by analyzing the independence relationships between all of the

optimizations in our compiler.

Thus, the major contributions of this thesis are:

1. This is the first research to analyze the optimization phase interactions to reduce the phase order search space and improve code quality.
2. We show that the problem of false register dependence is different from register pressure issues, and significantly impacts the size of the phase order search space.
3. We develop techniques to reduce false register dependence that substantially shrink the search space size and improve the quality of code generated by compiler optimizations.
4. We show that removing cleanup phases from the phase order search does not affect the quality of generated code, but significantly reduces the size of the search space.
5. We show that partitioning the optimization phase set into groups of mutually-independent phases and applying a multiple stage phase order search can reduce the search space to a fraction of its original size.
6. We identify criteria that may be used to automatically generate optimization set partitions for a multiple stage phase order search and demonstrate their use.

Chapter 2

Related Work

Issues revolving around optimization phase ordering and selection have been in the spotlight of compiler optimization research for over three decades, and as such, there is a wide body of existing research in this area. Older research studied the interactions between specific pairs of optimization phases. Leverett noted the interdependence between the phases of *constant folding* and *flow analysis*, and *register allocation* and *code generation* in the PQCC (Production-Quality Compiler-Compiler) project [23]. Vegdahl studied the interaction between *code generation* and *compaction* for a horizontal VLIW-like instruction format machine [26], and suggested various approaches to combine the two phases together for improved performance in certain situations. The interaction between *register allocation* and *code scheduling* has been studied by several researchers. Suggested approaches include using postpass scheduling (after *register allocation*) to avoid overusing registers [8, 13], and methods to combine the two phases into a single pass [9]. Earlier research has also studied the interaction between *register allocation* and *instruction selection* [3], and suggested using a common representation language for all the phases of a compiler, allowing them to be re-invoked repeat-

edly to take care of several such phase re-ordering issues. Most of these research works studied, what we call, *true* phase interactions between pairs of optimization phases. On the other hand, we focus on eliminating *false* phase interactions that exist due to naive implementation of optimization phases in the entire compiler backend.

Strategies to address the phase ordering problem generally pursue one of two paths: a model-driven approach or an empirical approach. A model-driven or analytical approach attempts to determine the properties of optimization phases, and then use some of these properties at compile time to decide what phases to apply and how to apply each phase. Some such studies use static and dynamic techniques to determine the enabling and disabling interactions between optimization phases. Such observations allow researchers to construct a single *compromise* phase ordering offline [27, 28] or generate a *batch* compiler that can automatically adapt its phase ordering at runtime for each application [18]. However, none of these works made any attempt to understand the causes behind those phase interactions. Follow-up work on the same topic has seen the use of additional analytical models, including code context and resource (such as cache) models, to determine and predict other properties of optimization phases such as the *impact* [29], and the *profitability* of optimizations [30]. Even after substantial progress, the fact remains that properties of optimization phases, as well as the relations between them are, as yet, poorly understood, and model-driven approaches find it hard to predict the best phase ordering in most cases.

With the growth in computational power, the most popular methods to resolve all optimization application issues involve *empirical* approaches. Researchers have observed that *exhaustive* evaluation of the phase order search space to find

the optimal function/program instance, even when feasible, is generally too time-consuming to be practical. Therefore, most research in addressing phase ordering employs iterative compilation to partially evaluate a part of the search space that is most likely to provide good solutions. Many such techniques use machine learning algorithms, such as genetic algorithms, hill-climbing, simulated annealing and predictive modeling to find effective (but, potentially suboptimal) optimization phase sequences [1, 2, 7, 14, 15, 20, 21]. Other approaches employ statistical techniques such as fractional factorial design and the Mann-Whitney test to find the set of optimization flags that produce more efficient output code [4, 12, 24]. Researchers have also observed that when expending similar effort most heuristic algorithms produce comparable quality code [2, 21]. The results presented in this thesis can enable iterative searches to operate in smaller search spaces, allowing faster and more effective phase sequence solutions.

Investigators have also developed algorithms to manage the search time during iterative searches. Static estimation techniques have been employed to avoid expensive program simulations for performance evaluation [6, 22, 25]. Agakov et al. characterized programs using static *features* and developed adaptive mechanisms using statistical correlation models to reduce the number of sequences evaluated during the search [1]. Kulkarni et al. employed several pruning techniques to detect *redundant* phase orderings to avoid over 84% of program executions during their genetic algorithm search [17]. However, in contrast to our approach, none of these methods make any attempt to understand phase interactions and alter the actual search space itself.

Research has also been conducted to completely enumerate and explore components of the phase order search space. In the next chapter, we describe our

earlier work that used several novel concepts enabling us to exhaustively evaluate the entire optimization phase order search space over all available phases in our compiler for most functions in our embedded systems benchmarks [18, 19, 22]. Some other work has attempted enumerations of search spaces over a small subset of available optimizations [2]. Each of these enumerations typically required several processor months even for small programs. Most of these research efforts have found the search space to be highly non-linear, but with many local minima that are close to the global minimum [2, 15, 21]. Such analysis has helped researchers devise better heuristic search algorithms. We believe that our present work to understand and reduce the phase order search space will further benefit all such exhaustive enumeration schemes.

Chapter 3

Background

In this chapter we provide an overview of the phase ordering/selection problems, and define some terminology. This overview will be followed by a description of our interpretation of this problem, and an overview of the algorithm we use to enumerate the phase order search space.

3.1 *Attempted* Optimization Phase Order Space

In its most basic form, the definition of the problem of optimization phase ordering can be very broad. Any possible combination and ordering of optimizations of *unbounded* length, including unrestricted phase repetitions, can constitute valid sequences of optimization phases. For example, considering three optimization phases, `a`, `b`, and `c`, the search space will include points corresponding to the sequences shown in Table 3.1.

Thus, such a naïve interpretation of the phase ordering problem can, indeed, lead to a huge search space. The possibility of optimization phases *enabling* each other results in phases being active multiple times in the same sequence. Conse-

Attempted Search Space		
<no opts>	a c b	b a
a	b a c	c a
b	b c a	c b
c	c a b	a b c a b c
a b	c b a	a b a c b a b a ...
a c	a b a	c c c a b b a c ...
b c	b b c	....
a b c	.....	....

Table 3.1. The unbounded attempted phase order space for three optimization phases, *a*, *b*, and *c*. Phases colored in red are sub-categorized in the phase selection search space. Phases colored in green are sub-categorized in the phase ordering search space. Sequence a b c belongs to both sub-categories.

quently, the naive optimization phase order search space is virtually unbounded.

3.2 Categorization of the Phase Ordering Problem

Clearly, searching for a good optimization sequence in the naive phase order search space will be overly complicated, time-consuming, and, in theory, can proceed ad infinitum. To simplify common search strategies, most existing research efforts divide the problem (and by consequence, the search space) into two sub-categories by placing restrictions on what sequences are considered.

Phase Selection Problem : Phase sequences are not allowed to re-order optimization phases. Thus, a default phase order is considered. Phases can be turned on or off, mirroring the compile-time flags provided with several conventional compilers, such as gcc [10]. Phase sequences colored in red in Table 3.1 will fall under this categorization.

Phase Ordering Problem : The simplified view of the phase ordering problem assumes a default sequence length, either greater [7,20] or smaller [1,2] than

the number of distinct optimization phases in the compiler. Recognizing the possibility of phases enabling each other, most approaches allow arbitrary repetition of phases in each sequence. Thus, the phase sequences colored in green in Table 3.1 will fall under this categorization for a sequence length of three.

Even after the division of the basic phase ordering problem, each sub-problem still involves huge search spaces considering the number of optimizations typically present in current compilers (60 in gcc [14], 82 in SUIF [1]). Evaluating all these search points for each program seems impossible, or at least extremely hard and overly time-consuming. Researchers, therefore, use novel search strategies along with probabilistic and machine-learning heuristics to (intelligently) evaluate only a portion of the search space, and find a good per-application-specific phase sequence.

3.3 Our Approach

For this work, we will use the same technique described by Kulkarni et al. [22] to enumerate the phase order search space. Most approaches to address the phase ordering and phase selection problems attempt to evaluate the performance of code generated by different optimization phase sequences without accounting for the fact that many such sequences may produce the same code (function instance). Another way of interpreting the phase ordering/selection problem is to enumerate all possible function instances that can be produced by any combination of optimization phases for any possible sequence length. This interpretation of the phase ordering problem allows us to view the phase ordering search space as a directed acyclic graph (DAG) of *distinct* function instances. Each DAG is function or pro-

gram specific, and may be represented as in Figure 3.1 for a hypothetical program and for the three optimization phases, `a`, `b`, and `c`. Nodes in the DAG represent function instances, and edges represent transition from one function instance to another on application of an optimization phase. The unoptimized function instance is at the root. Each successive level of function instances is produced by applying all possible phases to the distinct nodes at the preceding level. It is assumed in Figure 3.1 that no phase can be successful multiple times consecutively without any intervening phase(s) in between. The algorithm terminates when no additional phase is successful in creating a new distinct function instance.

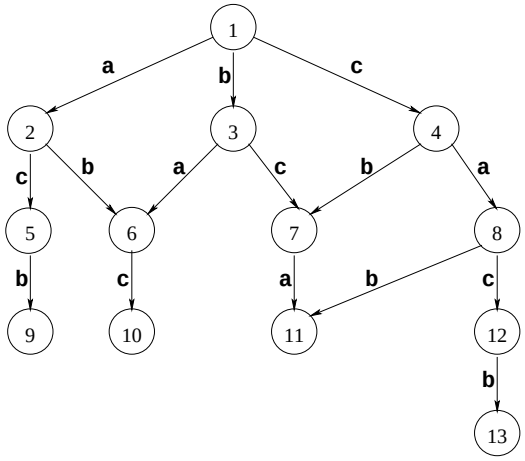


Figure 3.1. Optimization phase order search space DAG for a hypothetical program and compiler with three phases

Thus, our approach to addressing the phase application problem: (1) tackles the problem in its most basic form with no bounds or restrictions on the optimization search space, (2) subsumes the phase selection and phase ordering problems, both of which were earlier individually considered intractable, and (3) most importantly, can make it possible to generate/evaluate the entire search space, and determine the *optimal* function instance. Thus, any phase sequence from Table 3.1

can be mapped to a node in the DAG of Figure 3.1. This space of all possible *distinct function instances* for each function/program is, what we call, the *actual* optimization phase order search space.

Chapter 4

Experimental Setup

4.1 The VPO Compiler

The work in this thesis uses the Very Portable Optimizer (VPO) [3], which was a part of the DARPA and NSF co-sponsored National Compiler Infrastructure project. VPO is a compiler back end that performs all its optimizations on a single low-level intermediate representation called RTLs (Register Transfer Lists). Since VPO uses a single representation, it can apply most analysis and optimization phases repeatedly and in an arbitrary order. VPO compiles and optimizes one function at a time. This is important for the current study since restricting the phase ordering problem to a single function, instead of the entire file, helps to make the optimization phase order space more manageable. VPO has been targeted to produce code for a variety of different architectures. For this study we used the compiler to generate code for the StrongARM SA-100 processor using Linux as its operating system.

4.2 Optimization Space Details

Table 4.1 describes each of the 15 *optional* code-improving phases that we used during our exhaustive exploration of the optimization phase order search space. In addition, VPO also employs two compulsory phases, *register assignment* and *fix entry-exit*, that must be performed. *Register assignment* assigns pseudo registers to hardware registers.[1] In our experiments VPO implicitly performs register assignment before the first code-improving phase in a sequence that requires it. *Fix entry-exit* calculates required stack space, local/argument offsets, and generates instructions to manage the activation record of the runtime stack. The compiler applies *fix entry-exit* after performing the last optional code-improving phase in a sequence.

Two other optimizations, *merge basic blocks* and *eliminate empty blocks*, were removed from the optional optimization list used for the exhaustive search since these optimizations only change the internal control-flow representation as seen by the compiler, do not touch any instructions, and, thus, do not directly affect the final generated code. These optimizations are now implicitly performed after any transformation that has the potential of enabling them. Finally, after applying *fix entry-exit*, the compiler also performs predication and instruction scheduling before the final assembly code is produced. These last two optimizations should be performed late in VPO's compilation process, and so are not included in the set of phases used for exhaustive optimization space enumeration.

A few dependences between some optimization phases in VPO makes it illegal for them to be performed at certain points in the optimization sequence. The

[1] In VPO, pseudo registers only represent temporary values and not variables. Before register allocation, all program variables are assigned space on the stack.

Optimization Phase	Code	Description
branch chaining	b	Replaces a branch/jump target with the target of the last jump in the chain.
common subexpression elimination	c	Performs global analysis to eliminate fully redundant calculations, which also includes global constant and copy propagation.
dead code elimination	d	Removes basic blocks that cannot be reached from the function entry block.
loop unrolling	g	To potentially reduce the number of comparisons and branches at run time and to aid scheduling at the cost of code size increase.
dead assignment elimination	h	Uses global analysis to remove assignments when the assigned value is never used.
block reordering	i	Removes a jump by reordering blocks when the target of the jump has only a single predecessor.
loop jump minimization	j	Removes a jump associated with a loop by duplicating a portion of the loop.
register allocation	k	Uses graph coloring to replace references to a variable within a live range with a register.
loop transformations	l	Performs loop-invariant code motion, recurrence elimination, loop strength reduction, and induction variable elimination on each loop ordered by loop nesting level.
code abstraction	n	Performs cross-jumping and code-hoisting to move identical instructions from basic blocks to their common predecessor or successor.
evaluation order determination	o	Reorders instructions within a single basic block in an attempt to use fewer registers.
strength reduction	q	Replaces an expensive instruction with one or more cheaper ones. For this version of the compiler, this means changing a multiply by a constant into a series of shift, adds, and subtracts.
branch reversal	r	Removes an unconditional jump by reversing a conditional branch when it branches over the jump.
instruction selection	s	Combines pairs or triples of instructions that are are linked by set/use dependencies. Also performs constant folding.
useless jump removal	u	Removes jumps and branches whose target is the following positional block.

Table 4.1. Candidate Optimization Phases Along with their Designations

first restriction is that *evaluation order determination* can only be performed before *register assignment*. *Evaluation order determination* is meant to reduce the number of temporaries that *register assignment* later allocates to registers. Also, in some of our current experiments (presented in Chapter 5), *evaluation order determination* is always performed implicitly as part of *register assignment*. We do not believe this combination significantly affects the results of this study, but we removed this restriction in our later experiments. VPO also restricts some optimizations that analyze values in registers, such as *loop unrolling*, *loop strength reduction*, *induction variable elimination* and *recurrence elimination*, to be performed after *register allocation*. Many of these phases depend on the detection of basic induction variables and VPO requires these to be in registers before they are detected. These phases can be performed in any order after *register allocation* is applied. *Register allocation* itself can only be effective after *instruction selection* so that candidate load and store instructions can contain the addresses of arguments or local scalars. Finally, there are a set of phases that require the allocation of registers and must be performed after *register assignment*.

VPO is a compiler back end. Many other optimizations not performed by VPO, such as loop tiling/interchange, inlining, and some other interprocedural optimizations, are typically performed in a compiler frontend, and so are not present in VPO. We also do not perform ILP (frequent path) optimizations since the ARM architecture, our target for this study, is typically implemented as a single-issue processor and ILP transformations would be less beneficial. In addition, frequent path optimizations require a profile-driven compilation process that would complicate this study. In this study we are investigating only the phase ordering problem and do not vary parameters for how phases should be applied.

For instance, we do not attempt different configurations of loop unrolling, but always apply it with a loop unroll factor of two since we are generating code for an embedded processor where code size can be a significant issue.

It is important to realize that all optimization phases in VPO, except *loop unrolling* can be *successfully* applied only a limited number of times. Successful application of each phase depends on the presence of both the program inefficiency targeted by that phase, as well as the presence of architectural features required by the phase. Thus, (1) *register allocation* is limited (in the number of times it can be successfully applied) by the number of live ranges in each function. (2) *Loop invariant code motion* is limited by the number of instructions within loops. (3) *Loop strength reduction* converts regular induction variables to basic induction variables, and there are a limited number of regular induction variables. (4) There are a set of phases that eliminate jumps (*branch chaining, block reordering, loop jump minimization, branch reversal, useless jump removal*), and these are limited by the number of jumps in each function. (5) *Common subexpression elimination* is limited by the number of calculations in a function. (6) *Dead assignment elimination* is limited by the number of assignments. (7) *Instruction selection* combines instructions together and is limited by the number of instructions in a function. (8) *Induction variable elimination* is limited by the number of induction variables in a function. (9) *Recurrence elimination* removes unnecessary loads across loop iterations and is limited by the number of loads in a function. (10) *Code abstraction* is limited by the number of instructions in a function.

Loop unrolling is an optimization that can be attempted an arbitrary number of times, and can produce a new function instance every time. We restrict loop unrolling to be attempted only once for each loop. This is similar to the restriction

placed on loop unrolling in most compilers. Additionally, optimizations in VPO never undo the changes made by another phase. Even if they did, our approach could handle this since the function instance graph (explained in Section 3.3) would no longer be a DAG, and would contain cycles. Thus, for any function, the number of distinct function instances that can be produced by any possible phase ordering of any (unbounded) length is finite, and exhaustive search to enumerate all function instances should terminate in every case.

4.3 Our Benchmark Set

For these experiments we used a subset of the benchmarks from the *MiBench* benchmark suite, which are C applications targeting specific areas of the embedded market [11]. We selected two benchmarks from each of the six categories of applications in MiBench. Table 4.2 contains descriptions of these programs. The first two columns in Figure 4.2 show the benchmarks we selected from each application category in MiBench. The next column displays the number of lines of C source code per program, and the last column in Figure 4.2 provides a short description of each selected benchmark. VPO compiles and optimizes individual functions at a time. The 12 benchmarks selected contained a total of 246 functions, out of which 86 were executed with the input data provided with each benchmark.

4.4 Setup for Search Space Exploration

Our goal in this research is to understand optimization phase interactions and their effect on the size of the phase order search space. Therefore, we implement

Category	Program	#Lines	Description
auto	bitcount	584	test processor bit manipulation abilities
	qsort	45	sort strings using the quicksort sorting algorithm
network	dijkstra	172	Dijkstra's shortest path algorithm
	patricia	538	construct patricia trie for IP traffic
telecomm	fft	331	fast fourier transform
	adpcm	281	compress 16-bit linear PCM samples to 4-bit samples
consumer	jpeg	3575	image compression and decompression
	tiff2bw	401	convert color *tiff* image to b&w image
security	sha	241	secure hash algorithm
	blowfish	97	symmetric block cipher with variable length key
office	string-search	3037	searches for given words in phrases
	ispell	8088	fast spelling checker

Table 4.2. MiBench Benchmarks Used in the Experiments

the framework to generate exhaustive per-function phase order search spaces using all of VPO's 15 reorderable optimization phases, as proposed by Kulkarni et al. [22]. Redundancy detection employed by this algorithm enables us to prune away significant portions of the phase order search space, and allows exhaustive search space enumeration for most (96%) of the functions in our benchmark suite with the default compiler configuration.

As mentioned above, our compiler generates code for the StrongARM SA-100 processor using Linux as its operating system. Even though native execution of the benchmarks on the ARM system to measure dynamic runtime performance would be ideal, we were not able to do so due to resource constraints. Mainly, we did not have access to an ARM machine that runs Linux, and which also supports our compilation framework. Secondly, ARM machines are considerably slower than state-of-the-art x86 machines, so performing hundreds of long-running ex-

periments will require a significant number of custom ARM machines, which was infeasible for us to arrange. Therefore, we used the SimpleScalar set of functional and cycle-accurate simulators [5] for the ARM to get dynamic performance measures.

Invoking the *cycle-accurate* simulator for evaluating the performance of every distinct phase sequence produced by the search algorithm is prohibitively expensive. Therefore, we have adopted another technique that can provide quick *dynamic instruction counts* for all function instances with only a few program simulations per phase order search [6, 22]. In this scheme, program simulation is only needed on generating a function instance with a yet unseen *control-flow* to determine the number of times each basic block in that control-flow is reached during execution. Then, dynamic performance is calculated as the sum of the products of each block's execution count times the number of static instructions in that block. Interestingly, researchers have also shown that dynamic instruction counts bear a strong correlation with simulator cycles for simple embedded processors [22]. Note also, that the primary goal of this work is to uncover further redundancy in the phase order search space and reduce the time for phase order searches, while still producing the original best phase ordering code. Our dynamic counts are primarily used to validate such performance comparisons.

4.5 The Testing Framework

As this study developed, and we continued to add more benchmark functions, new compiler configurations, and more sophisticated data processing, we eventually built a centralized testing framework to manage our experiments. We modified VPO to accept a configuration file specifying different VPO configuration

options (e.g. what type of experiment to run) as well as which function in which benchmark to compile. We wrote scripts to automate generation of the VPO configuration files, as well as scripts to aggregate and analyze the resultant data files. We also have access to a high performance computing cluster to run our compute-intensive experiments. The Bioinformatics Cluster at the Information and Telecommunication Technology Center (ITTC) at the University of Kansas contains 176 nodes (with 4GB to 16GB of main memory on each node) and 768 total processors (with frequencies ranging from 2.8GHz to 3.2GHz). With this computing power combined with our testing framework, we were able to parallelize the phase order searches by running many different searches on individual nodes of the cluster. Each set of experiments can be completely described within a configuration file, and our python-based scripts are able to start many simultaneous search space exploration runs for each configuration with only a few short commands. Moreover, this framework also allows us to check the status of runs on different cluster nodes, stop and restart experiments on the fly, and automatically accumulate and analyze the results after finishing each set of runs.

Despite the available computing power, enumerating the phase order search space for some functions was still not possible. Search space enumerations that took longer than two weeks to gather or that generated raw data files larger than the maximum allowed on our 32-bit system (2.1GB) were simply stopped and discarded.

Chapter 5

False Phase Interactions

Architectural registers are a key resource whose availability, or the lack thereof, can affect (enable or disable) several compiler optimization phases. It is well-known that the limited number of available registers in current machines and the requirement for particular program values (like arguments) to be held in specific registers hampers compiler optimizations and is a primary cause for the phase ordering problem [3]. In this chapter, we describe and explain our findings regarding the effect of register availability and assignment on phase interactions, and the impact of such interactions on the size of the phase order search space.

Towards this goal, we employed the algorithm outlined in Section 3.3 to generate the exhaustive phase order search spaces for a few of our benchmark functions. We also designed several scripts to assist our manual study of these search spaces to detect and analyze the most common phase interactions. Surprisingly, we observed that many individual phase interactions occur, not due to conflicts caused by limited number of available registers, but by the particular register *numbers* that are used in surrounding instructions. Different phase orderings can assign different registers to the same program live ranges. These different register assign-

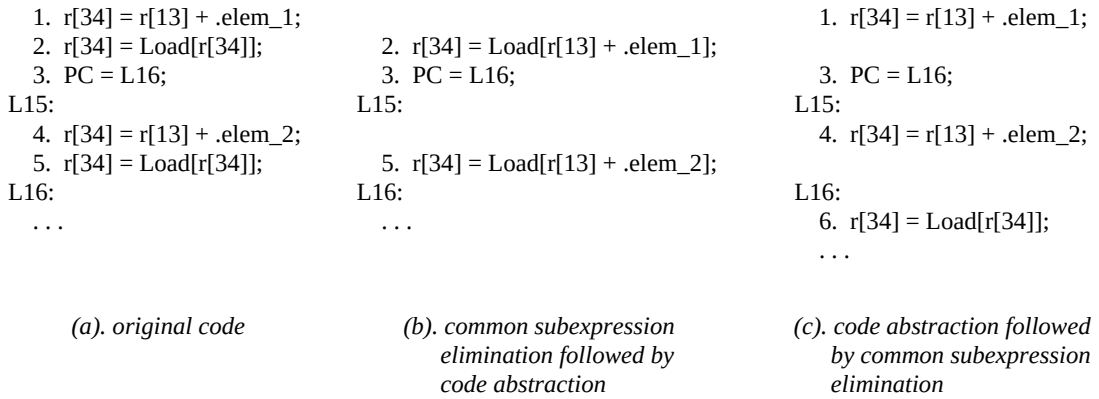


Figure 5.1. True phase interaction between *common subexpression elimination* and *code abstraction*

ments sometimes result in false register dependences that disable optimization opportunities for some phase orderings while not for others, and cause optimizations applied in different orders to produce distinct codes. We call phase interactions that are caused by false register dependences as *false interactions*. Such false interactions are often quite arbitrary and not only impact the search space size, but also make it more difficult for manual and intelligent heuristic search strategies to *predict* good phase orderings.

5.1 Examples of Phase Interactions

Figure 5.1 shows an example of a *true* phase interaction between *common subexpression elimination (CSE)* and *code abstraction*.[1] Figure 5.1(a) shows the code before applying either of these two phases. Figures 5.1(b) and 5.1(c) show code instances that are produced by applying CSE and code abstraction in different orders. Applying CSE to the code in Figure 5.1(a) propagates the values stored in `r[34]` in instructions 1 & 4 forward to instructions 2 & 5 to produce

[1]The description of these phases can be found in Table 4.1.

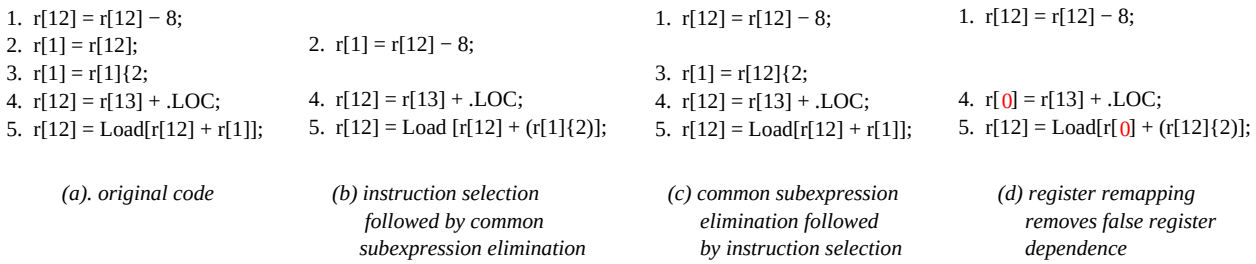


Figure 5.2. *Register remapping* eliminates false register dependence

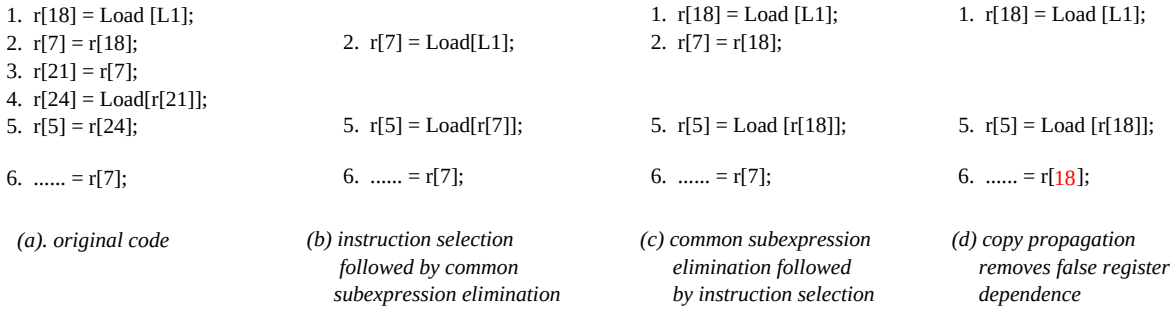


Figure 5.3. *Copy propagation* eliminates false register dependence

the code shown in Figure 5.1(b). However, applying code abstraction to the code in Figure 5.1(a) (before CSE) identifies the duplicate `r[34] = Load[r[34]];` instructions as an opportunity for cross jumping and moves this instruction into the block labeled `L16`. This transformation disables CSE and the result is the inferior code shown in Figure 5.1(c). This is an example of a *true* phase interaction because it arises from conflicting optimization strategies.

Figures 5.2 and 5.3 illustrate examples of phase interactions between *instruction selection* and *CSE* due to false register dependence. In the first example, we can see that the code in Figure 5.2(c) is inferior due to the reuse of register `r[12]`, which prevents instruction selection (applied after CSE) from combining instructions numbered 3 & 5, and thus leaving an additional instruction in the generated code. Applying instruction selection before CSE avoids this false reg-

ister dependence issue, producing better code in Figure 5.2(b). Similarly, in the second example shown in Figure 5.3, applying CSE before instruction selection leaves a redundant copy instruction in the code (Figure 5.3(c)) due to an unfavorable register assignment. Even later and repeated application of optimization phases are often not able to correct the effects of such register assignments. Thus, phase interactions due to false register dependences can produce distinct function instances. Successive optimization phases working on such unique function instances produce even more distinct points in the search space in a cascading effect that often causes an explosion in the size of the phase order search space. In the next section, we describe our proposed solution for dealing with false register dependences.

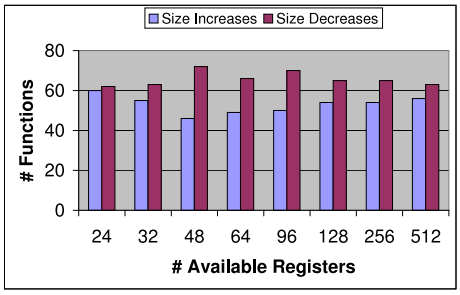
5.2 Effect of Register Pressure on Phase Order Space and Performance

We have seen that several optimization phase interactions are caused by different register assignments produced by different phase orderings. Such effects can cause a false register dependence to disable optimization opportunities for some phase orderings while not for others. False register dependence is often an artifact of the limited number of registers available on most machines. Such register scarcity forces optimization phases to be implemented in a fashion that reassigns the same registers often and as soon as they become available. If phases are implemented correctly, then a decrease in register pressure should also reduce false register dependences. If so, then we should expect the phase order search space to shrink with increasing register availability. However, a greater number of registers may also *enable* additional phase transformations, expanding the phase order

search space. In this section we present the first study of the effect of different numbers of available registers on the size of the phase order search space and the performance of the best code that is generated.

The ARM architecture provides 16 general-purpose registers, of which three are reserved by VPO (stack pointer, program counter, and link register). We modified the VPO compiler to produce code with several other register configurations ranging from 24 to 512 available registers. With the default VPO configuration, we are able to measure the phase order search space size for 236 (out of 246 total) benchmark functions. Given our processing time and speed limitations, we find the phase order search space for the remaining ten functions to be too vast to exhaustively explore. With our additional register configurations, there are two more functions that cannot be exhaustively evaluated because of size and / or time constraints. Thus, we measured and compared the search space size in all register configurations for 234 of our benchmark functions. Since the code generated by VPO with the other illegal register configurations cannot be simulated, we used a novel strategy to evaluate code performance in such cases. As described earlier in Section 4.4, measuring dynamic performance during our search space exploration only requires program simulations for instances with unseen basic block control-flows. Our performance evaluation strategy stores all the control-flow information generated for each function during its exhaustive search space search with 16 registers, and reuses that information to collect dynamic performance results during the other illegal VPO register configurations. We found that no additional control flows were generated for 73 of the 79 executed benchmark functions we were able to gather for these other VPO configurations. Thus, our scheme allows us to measure and compare the dynamic performance for 73 executed functions in all

(a) Search Space Size



(b) Functions with Size Differences

Figure 5.4. Search space sizes with different numbers of available registers

register configurations.

Figure 5.4(a) illustrates the impact of various register configurations on the size of the phase order search space, averaged over all 234 benchmark functions, as compared to the default search space size with 16 registers. Thus, we can see that the search space, on average, increases mildly with increasing number of available registers, and reaches a steady state when the additional registers are no longer able to create any further optimization opportunities for any benchmark functions. Figure 5.4(b) shows the number of functions that notice a difference in the size of the search space with changing number of available registers. Here, we see that there are typically more functions that see a search space size decrease as opposed to a search space size increase, while many functions do not notice any change in the size of their phase order search space. Performance for most of the 73 executed functions either improves or remains the same, resulting in an average improvement of 1.9% in all register configurations over the default.

The overall increase in the search space size indicates that the expansion caused by additional optimization opportunities generally exceeds the decrease (if any) caused by reduced phase interactions. In fact, we believe that the current im-

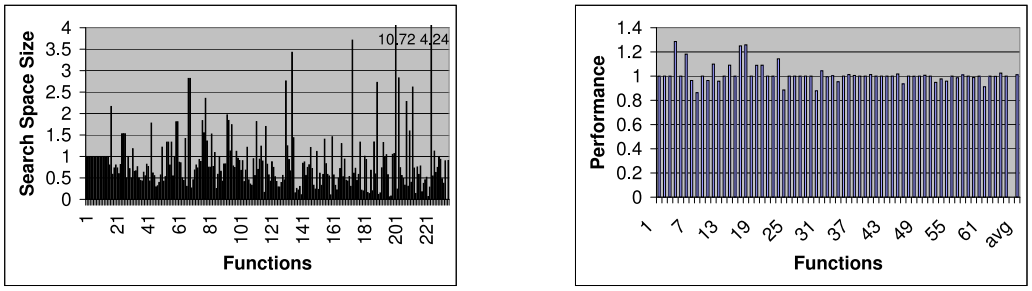
plementation of phases in VPO assumes limited registers and naturally reuses them whenever possible, regardless of register pressure. Therefore, limited number of registers is not the sole cause for false register dependences. Consequently, more informed optimization phase implementations may be able to minimize false register dependences and reduce the phase order search space. We explore this possibility further in the next two sections.

5.3 Measuring the Effect of False Register Dependence on the Phase Order Space

Our results in the previous section suggests that current implementation of optimization phases typically do not account for the effect of unfavorable register assignments producing false phase interactions. Rather than altering the implementation of all VPO optimization phases, we propose and implement two new transformations in VPO, *register remapping* and *copy propagation*, that are implicitly applied after every reorderable phase during our iterative search space algorithm to reduce false register dependences between phases. In this section, we show that removing such false phase interactions can indeed result in a dramatic reduction in the size of the phase order search space in a compiler configuration with sufficient (512) number of registers to avoid register pressure issues. In the next section we adapt and employ our techniques to reduce search space size and improve performance in the default ARM-VPO configuration with 16 registers.

5.3.1 Register Remapping to Reduce the Phase Order Search Space

Register *remapping* or *renaming* reassigns registers to live ranges in a function, and is a transformation commonly employed before *instruction scheduling* to re-



(a) Search Space Size

(b) Performance

Figure 5.5. Register-remapped configuration compared to default (512 registers). Functions are ordered corresponding to their search space size with the default VPO configuration (lower numbered functions correspond to smaller search spaces). The rightmost bar displays the average.

duce false register dependences and increase instruction level parallelism [8]. Figure 5.2(d) illustrates the effect of applying register remapping (after every phase) to the code in Figure 5.2(c) to remove the false interaction between instruction selection and CSE in Figure 5.2. In this study we use 512 available registers to remap as many of the conflicting live ranges as possible to unique register numbers. Figure 5.5(a) shows the effect of implicitly applying register remapping after every reorderable phase during the exhaustive search space exploration on the size of the search space for 233 benchmark functions[2] (the rightmost bar presents the average). Thus, our compiler configuration with implicit register remapping is able to reduce the search space size by 9.5% per function. Interestingly, this technique has more impact on functions with larger default search spaces. Thus, summing up the search space over all 233 functions, we find that the number of total distinct function instances reduces over 13% compared to the default.

Although register remapping cannot directly impact dynamic performance, it

[2]The search space for one of the original 234 functions with this configuration of VPO took too long to gather.

is an *enabling* phase that can provide more opportunities to optimizations following it. These new opportunities increase the size of the search space for several functions. Indeed, including register remapping as the 16^{th} reorderable phase in VPO causes an unmanageable increase in the size of the search space for all functions, preventing the exhaustive phase order searches for many functions from finishing even after several weeks. Therefore, it seems even more noteworthy that this transformation can reduce the search space size so substantially even as it enables more phases.

Of the 73 executed functions whose search spaces could be exhaustively evaluated for dynamic performance counts with the VPO configuration of 512 registers, 65 did not generate a new control flow with register remapping implicitly applied. Figure 5.5(b) shows the performance of these 65 functions when compared against the default VPO configuration with 512 registers. From this, we can see that register remapping only marginally affects the best code performance found during the exhaustive phase order search for most functions. For some functions, however, we found significant performance degradations with register remapping (up to 28.5% in one case). On average, performance degraded by 1.24%.

Detailed analysis of these performance degradations suggests that most of these issues stem from non-orthogonal use of registers and machine independent implementation of optimization phases in compilers. It is often the case that a system's ABI requires that certain values be held in specific registers at certain points in the code (e.g. arguments must be in specific registers before a function call). Such requirements may lead to performance variations when registers are remapped that would not exist if all registers were used orthogonally. With our ARM-Linux architecture, the first four registers (`r[0] - r[3]`) are used to hold

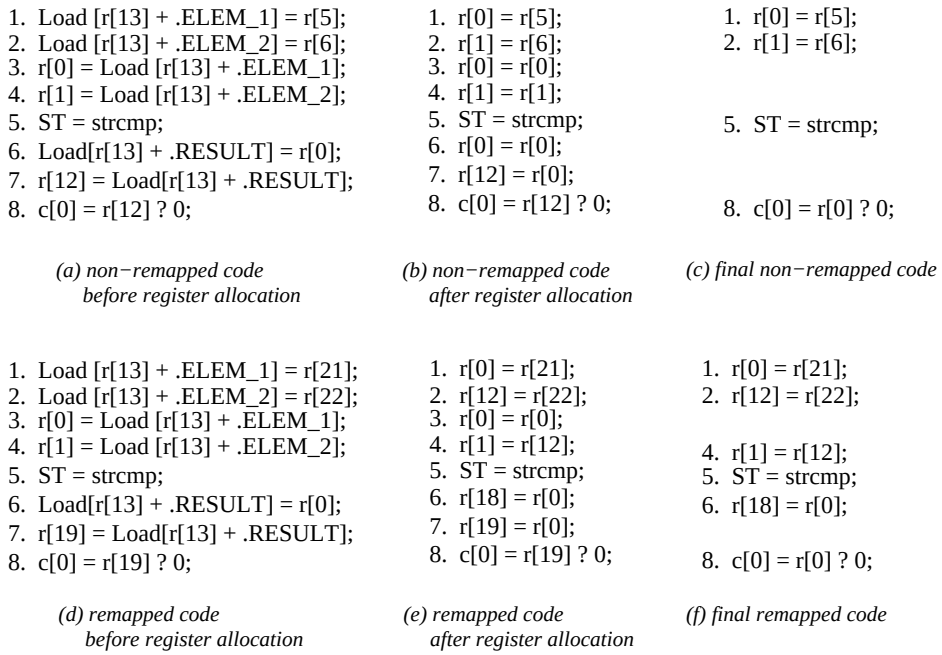


Figure 5.6. Non-orthogonal use of registers causes performance degradations in register-remapped code.

arguments to function calls and `r[0]` is used to store the return value. Compiler writers often design optimization phases to be unaware of such calling conventions so that they may be machine independent. In VPO, all of the reorderable phases in the phase order search are machine independent, and thus, do not account for the ARM-Linux calling convention. The epilogue phase *fix entry-exit*, which is applied immediately before dynamic instruction counts are recorded for each unique function instance in the phase order search, inserts or updates additional instructions when necessary to account for the target machine's calling convention.

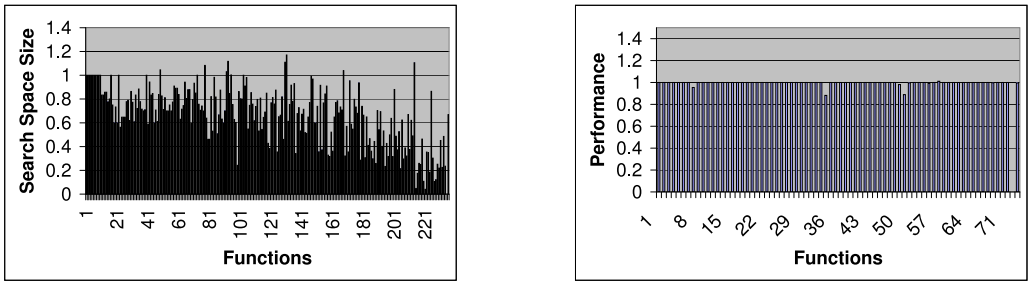
Figure 5.6 shows an example of how these requirements lead to performance degradations when register numbers are remapped after every optimization phase. Figures 5.6(a) - 5.6(c) show code generated by the default VPO configuration before and after register allocation and again after removing dead instructions. Fig-

ures 5.6(d) - 5.6(f) show the same sequence of function instances generated when register remapping is enabled. In the initial code, instructions 3, 4, and 6 are used to ensure the arguments and return value of the function call to *strcmp* are in the appropriate registers. In the non-remapped code, the register allocator happens to allocate these values to registers that make these instructions worthless, and thus, they are trivially removed. In the remapped code, however, remapping register numbers changes the order that registers are eventually allocated. Unaware of our target machine's calling convention, the register allocator allocates the function arguments and return value of the *strcmp* function call to registers that require instructions 4 and 6 in the final code. Intuitively, it seems these issues would only marginally affect a function's overall performance. However, in functions with only a few instructions or when these issues cause instructions inside frequently executed loops to persist, this can cause larger performance degradations.

5.3.2 Copy Propagation to Reduce the Phase Order Search Space

Next, based on our manual analysis of false phase interactions in VPO, we implemented *copy propagation* as another transformation to potentially further minimize the effects of unfavorable register assignments. Copy propagation is often used in compilers as a *clean-up* phase to remove copy instructions by replacing the occurrences of targets of direct assignments with their values. Figure 5.3(d) shows the result of applying copy propagation (after every phase) to the code in Figure 5.3(c), which results in code that is equivalent to that in Figure 5.3(b), thus negating the phase order issue originally present in this case between instruction selection and CSE.

We performed experiments to study the impact of implicitly applying copy

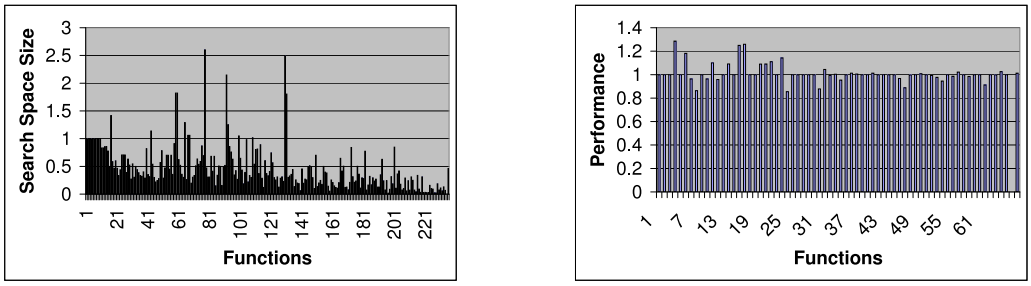


(a) Search Space Size

(b) Performance

Figure 5.7. Copy propagation configuration compared to default (512 registers). Functions are ordered corresponding to their search space size with the default VPO configuration (lower numbered functions correspond to smaller search spaces). The rightmost bar displays the average.

propagation to reduce false phase interactions on the size of the phase order search space. Figure 5.7(a) shows the change in the phase order search space size compared to default (with 512 registers) if every original VPO phase when successful is followed by the clean-up phase of copy propagation during exhaustive phase order space search for each function. Thus, the application of copy propagation is able to reduce the size of the search space by over 33%, on average. Furthermore, this technique also has a much more significant impact on functions with larger search spaces. Indeed, when we sum the search space size across all functions with this configuration and compare this to the sum of search space sizes with the default VPO configuration (with 512 registers), we find a total search space reduction of slightly more than 67%. Unlike the enabling effect produced by register remapping, copy propagation can directly improve performance by eliminating copy instructions. Using our earlier described technique to measure dynamic performance counts for a configuration with 512 registers, we were able to gather dynamic instruction counts for 72 benchmark functions. We found that applying copy propagation after every phase allows the exhaustive phase order searches



(a) Search Space Size

(b) Performance

Figure 5.8. Register remapping and copy propagation configuration compared to default (512 registers). Functions are ordered corresponding to their search space size with the default VPO configuration (lower numbered functions correspond to smaller search spaces). The rightmost bar displays the average.

to generate best function instances that achieve 0.41% better performance than default, on average. At the same time, we also observed that including copy propagation as a distinct (16^{th}) reorderable phase during the search space exploration (and not applying it implicitly after every phase) has a negligible effect on the quality of the code instances (performance improved by only 0.06% on average over the configuration with copy propagation implicitly applied). Moreover, such a VPO configuration almost doubles the size of the phase order search space with an increase of 98.8% over the default configuration, on average.

5.3.3 Combining Register Remapping and Copy Propagation

Interestingly, combining our two techniques is able to further reduce false register dependences and the size of the phase order search spaces. Thus, as shown in Figure 5.8(a), implicitly applying both register remapping and copy propagation after every phase reduces the size of the phase order search spaces by over 56.7%, per function, on average. This technique also has a much more significant effect on functions with larger search spaces. Thus, the total search space reduction when

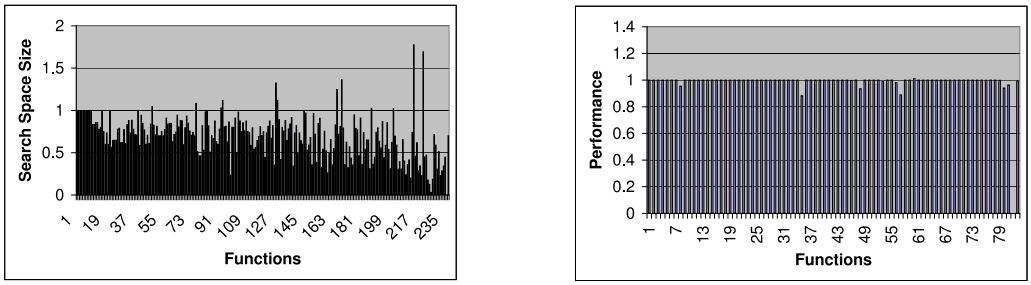
summed across all functions for this configuration compared to the default is an impressive 88.9%. We were able to gather performance data for 66 of our benchmark functions in this configuration. As can be seen in Figure 5.8(b), the effect on performance is similar to the effect of applying register remapping alone (Figure 5.5(b)), with the best average performance degrading by 1.24%. Since both our implicit phases reduce false register dependences, our results in this section demonstrate that false phase interactions caused by differing register assignments significantly contribute to the size of the phase order search space.

5.4 Eliminating False Register Dependence on Real Embedded Architectures

In the previous section, we showed that applying register remapping and copy propagation effectively reduce the phase order search space in a machine with virtually unlimited registers. Unfortunately, both these transformations show a tendency to increase register pressure, which can affect the operation of successive phases. In this section we show how we can employ our observations from the last section to adapt the behavior and application of these transformations for use on real embedded hardware to reduce search space size and improve generated code quality.

5.4.1 Reducing the Search Space with Copy Propagation

Aggressive application of copy propagation can increase register pressure and introduce register spill instructions. Increased register pressure can further affect other optimizations, that may ultimately result in changing the shape of the original phase order search space. For this reason, we develop a conservative



(a) Search Space Size

(b) Performance

Figure 5.9. Copy propagation configuration compared to default (16 registers). Functions are ordered corresponding to their search space size with the default VPO configuration (lower numbered functions correspond to smaller search spaces). The rightmost bar displays the average.

implementation of copy propagation that is only successful in cases where the copy instruction becomes redundant and can be removed later. Thus, our transformation only succeeds in instances where we can avoid increasing the register pressure.

We now apply our version of conservative copy propagation implicitly after each reorderable optimization phase during exhaustive phase order search space exploration (similar to its application in the last section). Figure 5.4.1 plots the size of the search space for each of our benchmark functions compared against the default VPO configuration. Thus, we can see that, similar to our results in the last section, our technique here reduces the size of the search space by 30% per function on average. Again, this technique tends to have more of an impact on functions with larger search spaces and the total search space size is reduced by 57.5% when summed across all functions. Implicit application of copy propagation during the exhaustive search algorithm improves the best generated code for a few functions, improving average performance by 0.56%.[3] We also found

[3]Because we were actually able to simulate each function instance, these results include all

that including copy propagation as a distinct (16^{th}) reorderable phase during the search space exploration on the real 16-register ARM machine gives similar results as on a hypothetical machine with 512 registers. The search space increases by over 133% per function on average and performance is only marginally better (0.19%) than implicit application.[4] Thus, prudent application of techniques to remove false register dependences can be very effective at reducing the size of the phase order search space on real machines.

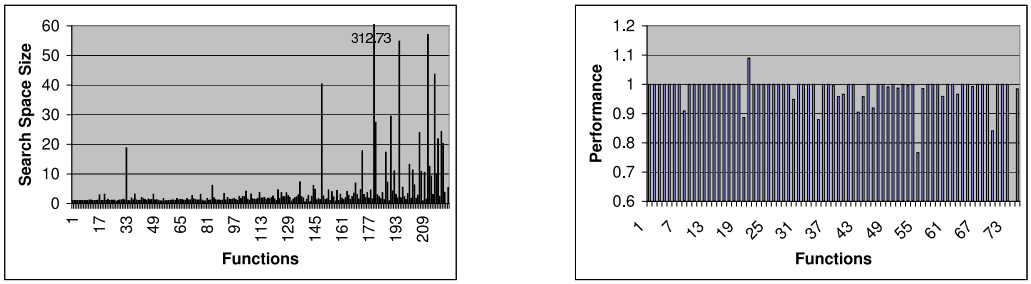
5.4.2 Improving Performance with Localized Register Remapping

We have not yet developed a similar conservative version of register remapping for implicit application during phase order searches. Instead, we employ register remapping to show how removing false register dependences *during* traditional optimization phases can be used to increase optimization opportunities and improve the quality of the generated code.

We select instruction selection to demonstrate our application of *localized* register remapping, but the same technique can also be applied to other phases. As illustrated in Figure 5.2(c), instruction selection (or some other optimization phase) might miss optimization opportunities due to some false register dependences. We modify instruction selection to only remap those live ranges that are blocking its application due to a false register dependence, if the transformation would be successful otherwise. Thus, when instruction selection fails to combine instructions due to one of more register conflicts, we identify the conflicting live ranges in these instructions, attempt to remap these so that they no longer conflict,

81 executed functions we were able to gather in the default configuration.

[4]Four of the 236 benchmark functions we gathered with the default configuration generated search spaces that were too large to gather with this configuration. Thus, these results include the remaining 232 benchmark functions, 79 of which were executed at least once.



(a) Search Space Size

(b) Performance

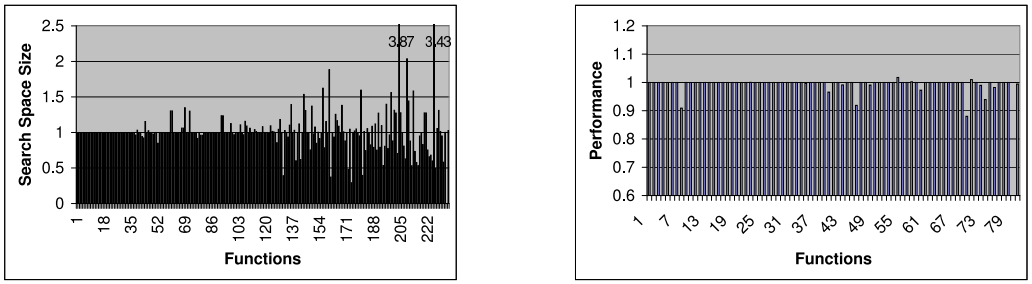
Figure 5.10. Aggressive localized register remapping configuration compared to default (16 registers). Functions are ordered corresponding to their search space size with the default VPO configuration (lower numbered functions correspond to smaller search spaces). The rightmost bar displays the average.

and then attempt to combine the instructions again. Such localized application of register remapping can minimize any increase in register pressure as well as potentially provide further optimization opportunities and generate better code.

In our first attempt at this technique, we found instruction selection may still fail after our localized register remapping due to some other issues (e.g. creating an invalid instruction). If such futile remappings are allowed to remain, many new (locally remapped) function instances may be introduced in the search space. This issue creates an explosion in the size of the search space for several of our benchmark functions. On average (as can be seen in Figure 5.10(a)), the search space size increased over 438% in the functions whose search spaces we were able to completely enumerate with this technique enabled.[5] Promisingly, as can be seen in Figure 5.10(b), we found that the best code improved by 1.51%, on average.

To eliminate this substantial increase in search space size, we implemented a more conservative form of localized register remapping that rolls back any failed

[5]The search spaces for 14 of the 236 functions we were able to completely enumerate with the default VPO configuration (8 of which were executed) became too large or took too long to gather with this configuration.



(a) Search Space Size

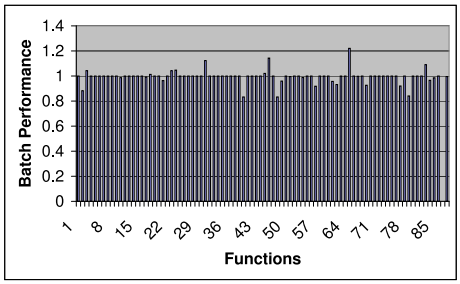
(b) Performance

Figure 5.11. Conservative localized register remapping configuration compared to default (16 registers). Functions are ordered corresponding to their search space size with the default VPO configuration (lower numbered functions correspond to smaller search spaces). The rightmost bar displays the average.

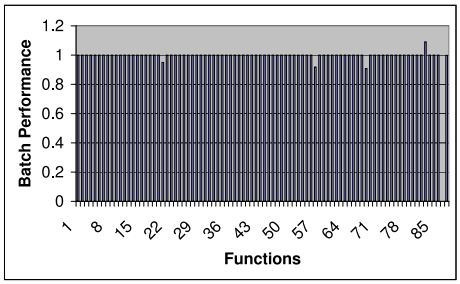
remappings in an attempt to negate any corresponding search space increases. Figures 5.11(a) and 5.11(b) show the search space and performance results for this configuration.[6] The search space size still increases, but only slightly, by 2.68% on average. The performance improves, but these improvements are more modest than the improvements seen with the more aggressive localized register remapping, only 0.55% on average, but with as much as 12% in the best case. This lower average improvement also indicates that at least some of the performance improvements seen in the aggressive configuration were due to register remapping interacting with optimization phases other than instruction selection.

Further, we tested the usefulness of this approach during the conventional (batch) compilation. The batch VPO compiler applies a fixed order of optimization phases in a loop until there are no additional changes made to the program by any phase. Figures 5.12(a) and 5.12(b) show the performance of the batch compiler with aggressive and conservative localized register remapping for each

[6]We were unable to completely enumerate 3 of the 236 function search spaces with this configuration, 2 of which were executed. Thus, these results show search spaces of 233 functions, and performances of 79 functions.



(a) Aggressive Localized Register Remapping



(b) Conservative Localized Register Remapping

Figure 5.12. Batch performance results for localized register remapping. The Functions are unordered. The rightmost bar displays the average.

of our 86 executed benchmark functions. We found that, on average, the aggressive configuration improved the batch compiler performance by 0.44%, while the conservative configuration improved it by 0.15%. The aggressive configuration has a greater impact on performance, but may actually degrade performance in some cases by introducing *new* register conflicts with other phases. Although such degradations may also occur in the conservative configuration, we did not observe this effect in the results for our benchmark functions.

The sole performance degradation in the conservative configuration was due to a conflict caused by remapping registers illustrated in Figure 5.13. Figures 5.13(a) - 5.13(d) show the code generated by the batch sequence with the default VPO configuration, and Figures 5.13(e) - 5.13(h) show the code generated by the same sequence with localized register remapping enabled. In this case, it is advantageous to hold the incremented values in these instructions in a common register (`r[4]`) because this allows the second application of instruction selection to combine the original code into only three instructions (as shown in Figure 5.13(d)). Remapping these incremented values to be held in several registers (which pro-

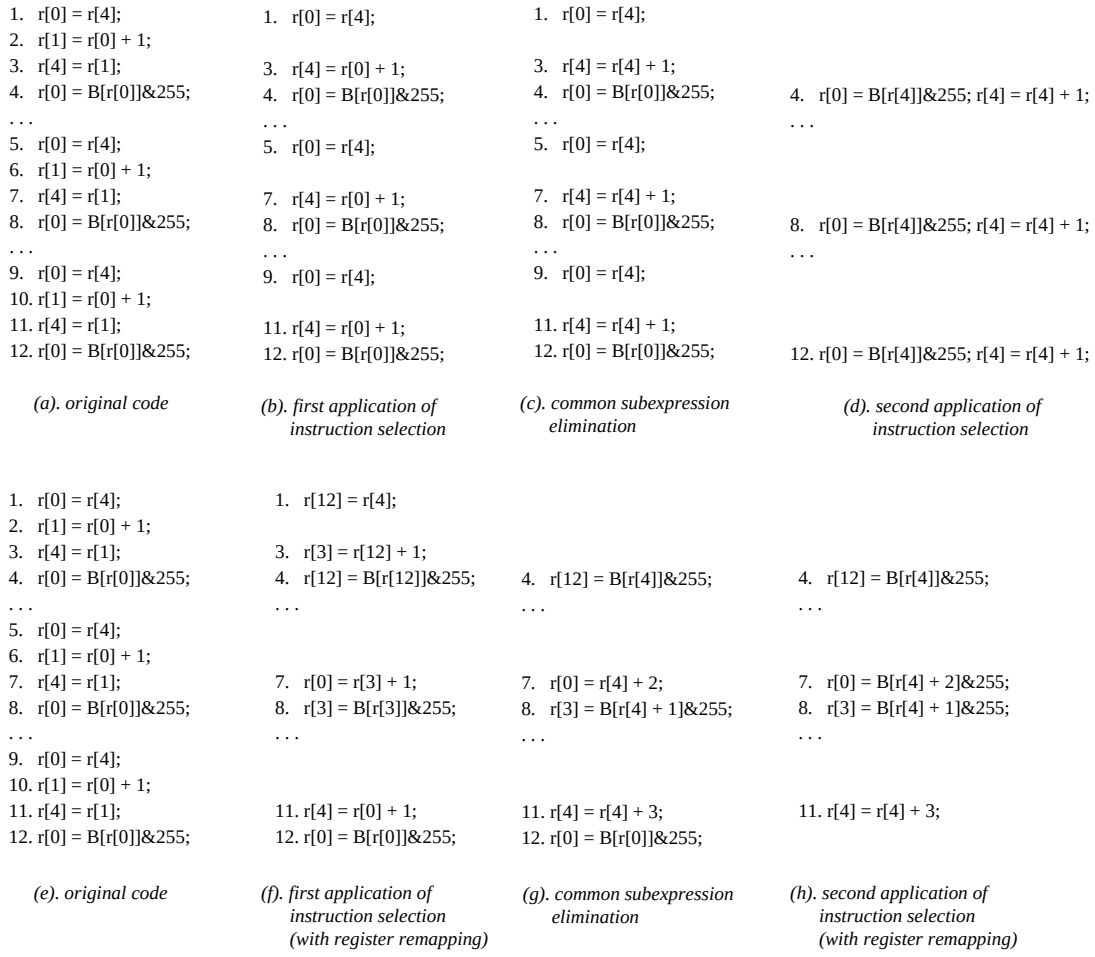


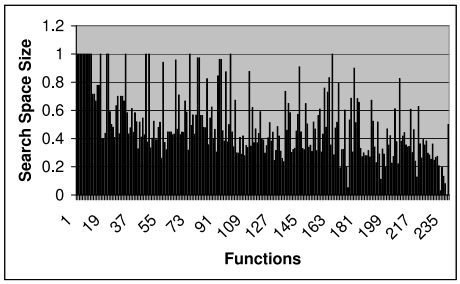
Figure 5.13. *Localized register remapping* may disable some optimization opportunities.

duces better code after the first application of instruction selection), prevents the second application of instruction selection from combining these instructions optimally. Thus, while register remapping typically eliminates false conflicts between phases, it may also disable optimization opportunities in certain situations.

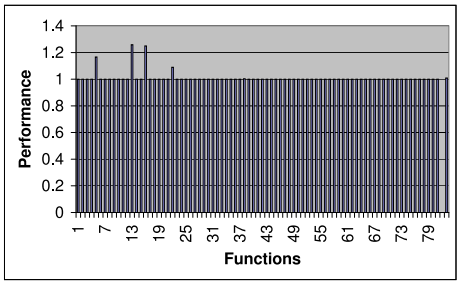
Chapter 6

Phase Independence

Two phases are *independent* of one another if applying them in different orders for any input code always leads to the same output code. If a phase is completely independent of all other phases, then that phase can be removed from the set employed during the exhaustive phase order search and applied implicitly after every relevant phase to reduce the phase order search space. We have observed that very few phases in VPO are completely independent of each other. However, several pairs of phases show none to very sparse phase interaction activity. Furthermore, some phases can be grouped together such that they only interact with other phases in that group, and do not interact with phases outside of that group. In this chapter, we show that reorganizing exhaustive phase order searches to exploit phase independence can drastically reduce the size of the search space while still generating code that performs as well as the code produced by the naïve phase order search. We investigate two complementary techniques to reorganize the phase order searches: eliminate cleanup phases from the phase order search, and multi-stage phase order searches with mutually independent sets of phases in each stage.



(a) Search Space Size



(b) Performance

Figure 6.1. Results of implicitly applying DAE after every relevant phase during the phase order search compared to default. Functions are ordered corresponding to their search space size with the default VPO configuration (lower numbered functions correspond to smaller search spaces). The rightmost bar displays the average.

6.1 Eliminating Cleanup Phases

Cleanup phases, such as *dead assignment elimination (DAE)* and *dead code elimination (DCE)* (see Table 4.1), do not consume any machine resources and merely assist other phases by cleaning junk instructions and blocks left behind by other code transforming optimizations. Such optimizations can be performed after any phase that might require cleanup. Thus, barring any false phase interactions, on-demand or implicit application of cleanup phases should not have any negative impacts on the performance of other optimization phases, but may result in a significant reduction in the space of phase orderings to explore.

We modified our phase order search algorithm to allow us to apply either DAE or DCE after every relevant phase. Figure 6.1(a) shows the effect of applying DAE implicitly on the search space size for each of our 236 benchmark functions. This configuration cuts the average search space size by just over 50%. It also has a much more significant impact on functions with larger default search spaces. Summing up the search space over all 236 functions, we find that the number of

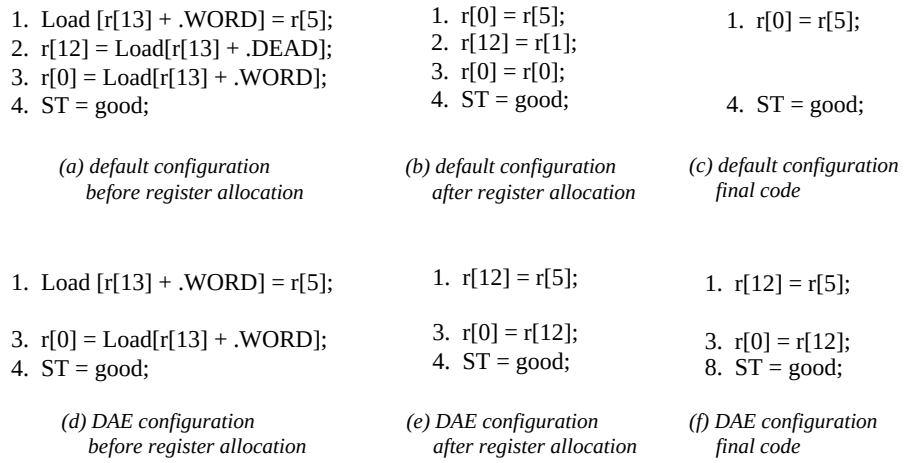


Figure 6.2. Non-orthogonal use of registers can cause performance degradations when DAE is implicitly applied after every phase.

total distinct function instances reduces by over 77% as compared to the default. Although this technique does not affect the best performance found for most functions, it does incur performance degradations in 5 of the 81 executed functions. These range from less than 0.4% up to 25.9%, with an average degradation across all functions of 0.95%.

Similar to what we saw with register remapping in Section 5.3.1, we found that most of these degradations stem from the non-orthogonal use of registers. Figure 6.2 shows an example. Figures 6.2(a) - 6.2(c) show code for our example function before and after register allocation and again after every other applicable phase has been applied in the default configuration. In this configuration, the dead assignment in instruction 2 is not removed until after register allocation is applied. In VPO, register allocation attempts to allocate registers one at a time in a fixed order. With the default ordering, the register allocator always attempts to allocate values to `r[12]` before allocating to `r[0]`. In Figure 6.2(b), we see that the register allocator allocates the value used in instruction 3 to `r[0]`

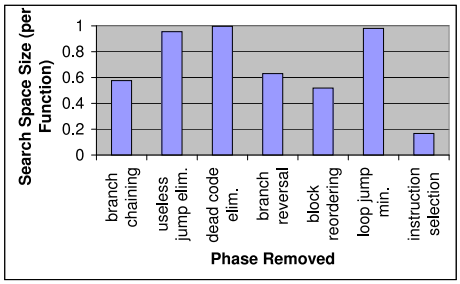
because allocating to `r[12]` would conflict with the assignment in instruction 2. This happens to be advantageous as now instruction 3 may be trivially removed as shown in Figure 6.2(c). Figures 6.2(d) - 6.2(f) show the code generated by applying the same sequence of phases in the configuration with DAE applied implicitly after every optimization phase. In this configuration, the dead assignment is removed before register allocation. Register allocation now chooses to allocate the value used in instruction 3 to `r[12]` (as allocating to this register no longer results in a conflict). Under this allocation scheme, instruction 3 must remain in the final code. In each of the examples we analyzed, we saw that these issues resulted in only one or two additional instructions in the final code. In spite of this, when these additional instructions appear in smaller functions or inside frequently executed loops, they may produce significant performance degradations.

We also tried applying the cleanup phase *dead code elimination (DCE)* after every relevant phase in the phase order search. DCE removes basic blocks that cannot be reached from the function entry block. In the default VPO configuration, opportunities to apply DCE are relatively rare. Moreover, the other optimizations in VPO already ignore unreachable code and are typically not affected by applying DCE. For these reasons, we did not expect removing DCE to have much of an impact on the search space. Indeed, we found that this technique reduced the search space in only one of our 236 benchmark functions (by 86%). This function was never executed, and none of the performances of the functions that were executed are affected by removing DCE from the search space.

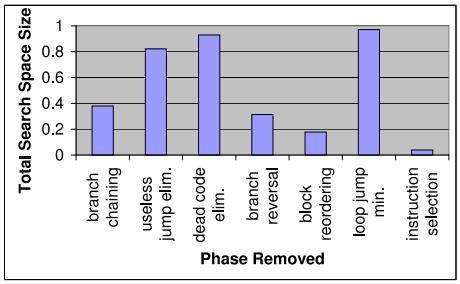
6.2 Eliminating Branch Optimizations

The set of optimization phases in many compilers can be naturally partitioned into two subsets: phases that affect the program control-flow and phases that depend on registers. Intuitively, *control-flow (branch) optimizations* should be naturally independent from non-branch optimizations. Using our knowledge of the VPO optimization set, we identified six of the fifteen reorderable VPO optimization phases as branch optimizations: *branch chaining*, *useless jump elimination*, *dead code elimination*, *branch reversal*, *block reordering*, and *loop jump minimization* (see Table 4.1 for a description of each of these). Using the same technique as we used with cleanup phases in Section 6.1, we modified VPO to apply each of these phases after every other relevant phase in the phase order search. We then enumerated the phase order search space for our benchmark functions with each of the branch optimizations removed (one at a time) from the search space. As a point of comparison, we also applied this same technique with the non-branch optimization, *instruction selection*.

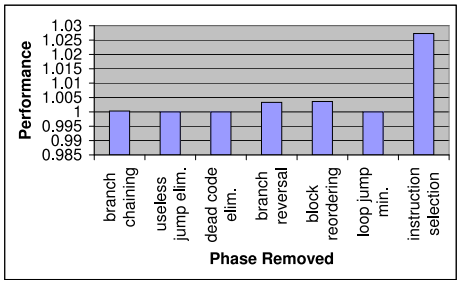
Figures 6.3(a) and 6.3(b) show the average search space size results for each of these configurations. We found that, in most cases, removing branch optimizations from the phase order search significantly reduces the search space size. Similar to what we saw in Section 6.1 with removing *dead assignment elimination* from the search space, removing the branch optimizations had more of an effect on functions with larger default search spaces. Figure 6.3(b) quantifies this effect by comparing the total number of distinct function instances summed over all 236 benchmark functions gathered in each configuration to the total number found in the default configuration. Removing instruction selection from the phase order search drastically reduces the average search space size by 83.3% per function and



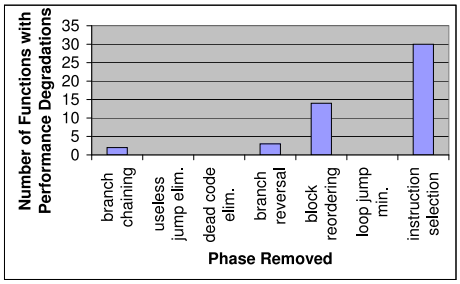
(a) Search Space Size (per Function)



(b) Total Search Space Size



(c) Average Performance Degradation (per Function)



(d) Number of Functions with Performance Degradations

Figure 6.3. Configurations with an optimization applied implicitly after every phase compared to the default configuration. In (a) and (b), the average across all 236 benchmark functions is shown. In (c) and (d), the results of the 81 executed benchmark functions are shown.

96.1% total (although, as we will see, this comes with a significant performance cost). Among the branch optimizations, removing block reordering from the phase order search yields the most significant reductions in search space size with an average reduction of 48.1% per function and an impressive 82.2% reduction in total search space size. Opportunities to apply *useless jump removal*, *dead code elimination*, and *loop jump minimization* are relatively rare compared to the other phases, and thus, removing these from the search space yield a smaller reduction in the search space size.

We also found that removing branch optimizations from the phase order search causes relatively few performance degradations. Figure 6.3(c) shows the average

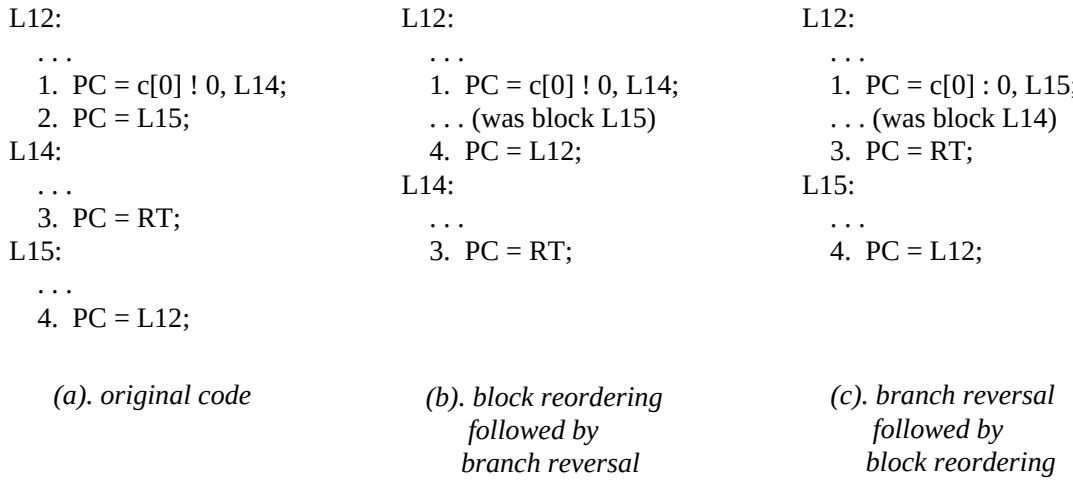


Figure 6.4. Interaction between block reordering and branch reversal. The operators '!' and ':' each compare two elements. '!' returns true if these are not equal. ':' returns true if these are equal.

performance results for each of these configurations compared to the default and Figure 6.3(d) shows the number of functions yielding a performance degradation in each configuration. Removing *instruction selection* produces the most significant impact on performance with 30 of the 81 executed functions experiencing some performance degradation and an average degradation of 2.73% per function. In the worst case, performance drops by more than 32.1% in this configuration. In contrast, removing branch optimizations from the search space has a much milder effect on performance. Some branch optimizations (specifically, *useless jump elimination*, *dead code elimination*, and *loop jump minimization*) may be removed without any noticeable impact on performance whatsoever. Among the impacting configurations, removing block reordering impacts the most functions (14), while removing branch reversal generates the largest performance degradation of a single function (24.4%). On average, removing block reordering has the largest per function performance degradation (of 0.36%).

Under our current understanding of the phase implementations in VPO, we

believe branch optimizations should not have many interactions with non-branch optimizations. It should be noted, however, that the sets of branch and non-branch optimizations may not always be disjoint. For example, in VPO, *loop unrolling* can modify the program control-flow and also requires registers. Despite this, detailed analysis of the performance degradations in these configurations suggests that most of these degradations are indeed due to interactions among the branch optimizations. Figure 6.4 shows an example of one such interaction between block reordering and branch reversal. Applying block reordering first to the original code, as shown in 6.4(b), removes the branch to `L15`, but also disables branch reversal. In contrast, applying branch reversal first (6.4(c)), removes the branch to `L14`, but disables block reordering and allows the branch to `L15` to persist. When this function is actually executed, the branch to `L15` is taken much more frequently than the branch to `L14`. Thus, this interaction causes a significant performance degradation when we remove branch reversal from the phase order search. Given this result, we next explore how to exploit mutually independent groups of phases in order to reduce the search space.

6.3 Multi-stage Phase Order Searches

In the previous section, we found that branch optimizations are largely independent from non-branch optimizations, but, in several cases, cannot be simply removed from the phase order search because of interactions with other branch optimizations. In this section, we show that a novel reorganization of the phase order search that exploits the independence relationship between these groups of phases drastically reduces the phase order search space without degrading performance.

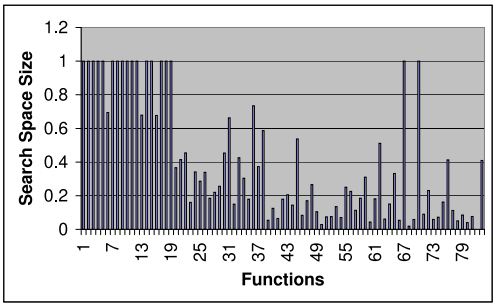


Figure 6.5. Multi-stage phase order search space size compared to default. Functions are ordered corresponding to their search space size with the default VPO configuration (lower numbered functions correspond to smaller search spaces). The rightmost bar displays the average.

6.3.1 The Multi-Stage Approach

Our new approach divides the phase order search into multiple searches conducted over different subsets of the optimization set. In the first stage, we conduct a phase order search using only the branch optimizations identified in the previous section as our optimization set. This will return the set of function instances that is reachable by applying any combination of the branch optimizations. Next, we search through these resulting function instances to find the best performing function instance(s). Finally, for each best performing function instance, we conduct another phase order search using the non-branch optimizations as our optimization set and with this function instance as a starting point.

We found that this multi-stage phase order search prunes the generated search space to a fraction of its original size without sacrificing performance. In our original configuration for the multi-stage phase order search, the optimization set in the second stage did not include any branch optimizations. We found that some functions may not reach an optimal performing function instance with this configuration because some of the non-branch optimizations may change the control

flow (by removing an instruction which results in the removal of a basic block) and enable one of the branch optimizations. When we include the branch optimizations in the optimization set of the second stage, we can generate at least one optimal performing function instance for each of the 81 executed benchmark functions while still pruning the search space significantly. Note that some functions have multiple optimal performing function instances in the default search space and this algorithm does not always produce all of them.

Figure 6.5 compares the search space gathered with this technique to the search space generated by the default configuration. We only compare search spaces generated for the 81 executed benchmark functions because this technique uses performance as a distinguishing criteria to prune the search space. On average, this technique reduces the original search space size by over 59% per function. This technique also has a much more significant impact on functions with larger default search spaces. We find that this technique reduces the total number of distinct function instances (summed across all 81 functions) by an impressive 88.4%.

6.3.2 Automatic Set Partitioning

The multi-stage phase order search described here requires intricate knowledge of the compiler's optimization set in order to partition this set into mutually independent subsets. Typically, compiler users do not know enough about their compiler to know which sets of optimizations might be mutually independent. Furthermore, although the branch / non-branch partitioning yields great search space reductions, there may exist optimization set partitionings which are able to reduce the search space even more within the multi-stage phase order search

```
{s}
    {s} -- 1928098  / 1930359  (99.88)  --> {b}
    {s} -- 1435397  / 1437547  (99.85)  --> {u}
    {s} -- 1865692  / 1871993  (99.66)  --> {r}
    {s} -- 2602782  / 2606233  (99.87)  --> {i}
    {s} -- 432      / 612      (70.59)  --> {o}
    {s} -- 19828    / 19876    (99.76)  --> {j}
    {s} -- 3619540  / 3622677  (99.91)  --> {h}
    {s} -- 531361   / 579346   (91.72)  --> {k}
    {s} -- 3518726  / 4373485  (80.46)  --> {c}
    {s} -- 521932   / 695560   (75.04)  --> {q}
    {s} -- 1322681  / 1428096  (92.62)  --> {l}
    {s} -- 1502813  / 1612770  (93.18)  --> {n}
    {s} -- 979361   / 1060829  (92.32)  --> {g}
```

Figure 6.6. Independence table for instruction selection. See Table 4.1 for the optimization corresponding to each code.

framework. Thus, we designed an algorithm to automatically partition the optimization set into multi-stage phase order search sets by analyzing independence statistics gathered from the default phase order search space.

In order to get a view of the independence relationships among the optimizations in our compiler, we first need to gather information about how often our optimization phases interact. In our case, we use the default phase order search space for each of the 81 executed benchmark functions as our data set. From this, we calculate an *independence score* for each pair of optimizations. This score is an estimation of how often each set of phases is independent. We calculate scores for each optimization pair by visiting all of the nodes in the input data. If two optimization phases are active at the same node (i.e. applying them leads to a new node), we increment the `total_active` value for this optimization pair. If applying these optimizations from this point in either order eventually leads to the same node, we increment the `total_independent` value for this optimization pair. The final independence score for each optimization pair is the ratio of its `total_independent` value to its `total_active` value.

We can construct independence tables summarizing the independence relation-

ships among our optimization phases. Figure 6.6 shows the independence table for *instruction selection*. A quick glance at this table shows that *instruction selection* is almost always independent from each of the branch optimizations, but interacts more often with some of the non-branch optimizations, especially *common subexpression elimination* and *strength reduction*. The independence score between *instruction selection* and *dead code elimination* is not listed because these two phases are never found to be active at the same node in our data set.

Using these tables, we can compute a set independence score between two optimization sets. The *optimization pair set* between two sets `A` and `B` is the set of all optimization pairs that can be formed by pairing an optimization from set `A` with an optimization from set `B`. Now, the *set independence score* between two sets `A` and `B` is simply the sum of the `total_independent` values between optimization pairs in `A` and `B`'s optimization pair set divided by the sum of the `total_active` values.

Next, we can use set independence scores to compute the most mutually independent subsets of our optimization set. For each possible subset of length $\lfloor \frac{n}{2} \rfloor$ (n being the total number of optimizations in our original set), compute the set independence score between this set and the set composed of the remaining optimizations. As set independence is a symmetric relationship, there is no need to compute set independence for subsets of length greater than $\lfloor \frac{n}{2} \rfloor$.

Finally, to choose the best partitions for the multi-stage phase order search, we choose the most mutually independent subsets that will also result in the largest search space reductions. Admittedly, we have no way of knowing which set partitions will most reduce the phase order search space. However, we have seen that search spaces grow exponentially as optimizations are introduced. Thus,

we seek to minimize the number of optimizations that will interact within one stage of the multi-stage phase order search. Thus, we choose the most mutually independent subset with length $\lfloor \frac{n}{2} \rfloor$ as the optimization set for the first stage of our multi-stage phase order search. This yields the subset:

$$\texttt{A=\{b, u, d, r, i, j, g\}}$$

Note that we have chosen the branch optimizations along with *loop unrolling*. *Loop unrolling* has a tendency to interact with the branch optimizations because it introduces new control flows. However, when *loop unrolling* is included in the first stage of the multi-stage phase order search, it is never active because it requires *register allocation* and *instruction selection* before it can be applied. Thus, using the optimization set chosen by our automatic set partitioning algorithm as the first stage in the multi-stage phase order search generates the same search space as using a branch / non-branch optimization partitioning.

Chapter 7

Future Work

There are several avenues for future work. In Chapter 5, we focused on phase interactions produced by false register dependences and different register assignments. In the future we plan to study other causes of false phase interactions and investigate possible solutions. One cause we believe is a conservative implementation of *instruction selection*, which misses some opportunities because it does not combine instructions when it cannot immediately improve the code. A more aggressive implementation of *instruction selection* may remove interactions such as those identified in the example in Figure 7.1. We believe that eliminating such false interactions will not only reduce the size of the phase order search space, but will also make the remaining interactions more predictable. We would like to explore if this predictability can allow heuristic search algorithms to detect better phase ordering sequences faster. In Section 5.4.2, we integrated localized register remapping with instruction selection to produce higher-quality code. In the future, we would like to similarly modify other compiler optimizations and study their effect on performance.

In Chapter 6, we exploited phase independence to reduce the search space size.

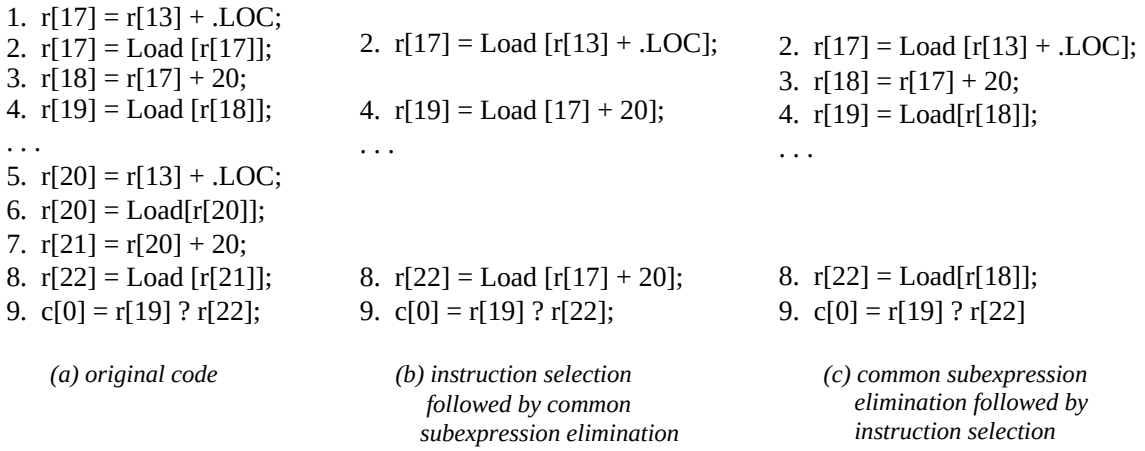


Figure 7.1. Phase interaction between *instruction selection* and *common subexpression elimination*. Instruction selection fails to remove instruction 3 in (c) because combining instructions 3 & 4 does not improve the final code (instruction 3 must remain due to the use of `r[18]` in instruction 8). A more aggressive implementation of instruction selection would combine instructions 3 & 4 (despite the fact that instruction 3 must remain), and, in a later pass, combine instructions 3 & 8, at which point instruction 3 would be removed, rendering (b) and (c) identical function instances.

In some cases, we found that interactions stemming from the non-orthogonal use of registers can cause performance degradations. We plan to modify VPO to account for the non-orthogonal use of registers and eliminate this as a cause of phase interaction. We believe heuristic (specifically, genetic) search algorithms can benefit from a multi-stage approach. We would like to compare the optimization independence relationships we found by analyzing the complete phase order search space with the relationships we can deduce from modified VPO configurations and heuristic search algorithms. Finally, we would like to refine our process of automatically partitioning the optimization set for the multi-stage phase order search (described in Section 6.3.2) and experiment with other optimization set partitions.

Chapter 8

Conclusions

Effectively addressing the optimization phase ordering problem is important for applications in the embedded systems domain. Unfortunately, in current compilers, most functions produce phase order search spaces that are infeasible or impractical to exhaustively explore. Thus, we develop several novel techniques for reducing the phase order search space.

We found that the problem of huge phase order search spaces is partly a result of the interactions between optimization phases that are caused by false register dependences. We also discover that due to the current implementation of optimization phases, even reducing the register pressure by increasing the number of available registers is not sufficient to eliminate false register dependences. Our new transformations, register remapping and copy propagation, to reduce false register dependences are able to substantially reduce the size of the phase order search spaces, but at the cost of increased register pressure that is not sustainable on real machines. We then showed that conservative implementation of these transformations during and between phases can still achieve impressive reductions in the search space size, while also achieving better code quality.

If an optimization phase is completely independent from all other phases in the phase order search space, we should be able to remove this phase from the search by applying it implicitly after every relevant phase. When we analyzed the independence relationships among the optimizations in our compiler, we found most phases only interact with a small group of other phases and these interactions tend to be sparse. We found removing cleanup phases, such as *dead assignment elimination* and *dead code elimination*, from the phase order search usually works very well, but causes significant performance degradations in a few cases. We found that branch and non-branch optimizations typically only interact among themselves. We showed that partitioning the optimization set into branch and non-branch optimizations and applying these in a staged fashion reduces the search space to a fraction of its original size without sacrificing performance. Finally, we described a method for automatically performing this partitioning without prior knowledge of the compiler's phase implementations.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, Washington, DC, USA, 2006.

[2] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 231–239, 2004.

[3] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 329–338, 1988.

[4] G. E. P. Box, W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building.* John Wiley & Sons, 1 edition, June 1978.

[5] D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[6] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Acme: adaptive compilation made efficient. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 69–77, 2005.

[7] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9, May 1999.

[8] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *Proceedings of the SIGPLAN '86 Conference on Programming Language Design and Implementation*, pages 11–16, June 1986.

[9] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 442–452, 1988.

[10] B. J. Gough. *An Introduction to GCC*. Network Theory Ltd., May 2005.

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

[12] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 123–132, Washington, DC, USA, 2005. IEEE Computer Society.

[13] J. L. Hennessy and T. Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, 1983.

[14] K. Hoste and L. Eeckhout. Cole: Compiler optimization level exploration. In *accepted in the International Symposium on Code Generation and Optimization (CGO 2008)*, 2008.

[15] T. Kisuki, P. Knijnenburg, , and M. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Internation Conference on Parallel Architectures and Compilation Techniques*, pages 237–246, 2000.

[16] T. Kisuki, P. Knijnenburg, M. O'Boyle, F. Bodin, , and H. Wijshoff. A feasibility study in iterative compilation. In *Proceedings of ISHPC'99, volume 1615 of Lecture Notes in Computer Science*, pages 121–132, 1999.

[17] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 171–182, Washington DC, USA, June 2004.

[18] P. Kulkarni, D. Whalley, G. Tyson, and J. Davidson. Exhaustive optimization phase order space exploration. In *Proceedings of the Fourth Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 306–308, March 26-29 2006.

[19] P. Kulkarni, D. Whalley, G. Tyson, and J. Davidson. In search of near-optimal optimization phase orderings. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers and tool support for embedded systems*, pages 83–92, 2006.

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

[21] P. A. Kulkarni, D. B. Whalley, and G. S. Tyson. Evaluating heuristic optimization phase order search algorithms. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 157–169, 2007.

[22] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson. Practical exhaustive optimization phase order exploration and evaluation. *ACM Transactions on Architecture and Code Optimization*, 6(1):1–36, 2009.

[23] B. W. Leverett, R. G. G. Cattell, S. O. Hobbs, J. M. Newcomer, A. H. Reiner, B. R. Schatz, and W. A. Wulf. An overview of the production-quality compiler-compiler

project. *Computer*, 13(8):38–49, 1980.

[24] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 319–332, 2006.

[25] S. Triantafyllis, M. Vachharajani, N. Vachharajani and D. I. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 204–215, 2003.

[26] S. R. Vegdahl. Phase coupling and constant generation in an optimizing microcode compiler. In *Proceedings of the 15th Annual Workshop on Microprogramming*, pages 125–133. IEEE Press, 1982.

[27] D. Whitfield and M. L. Soffa. An approach to ordering optimizing transformations. In *Proceedings of the second ACM SIGPLAN symposium on Principles & Practice of Parallel Programming*, pages 137–146, 1990.

[28] D. L. Whitfield and M. L. Soffa. An approach for exploring code improving transformations. *ACM Trans. Program. Lang. Syst.*, 19(6):1053–1084, 1997.

[29] M. Zhao, B. Childers, and M. L. Soffa. Predicting the impact of optimizations for embedded systems. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN Conference on Language, compiler, and tool for embedded systems*, pages 1–11, New York, NY, USA, 2003. ACM Press.

[30] M. Zhao, B. R. Childers, and M. L. Soffa. A model-based framework: An approach for profit-driven optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 317–327, Washington, DC, USA, 2005.