THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

FAST AND EFFECTIVE SOLUTIONS TO THE PHASE ORDERING PROBLEM

By

PRASAD A. KULKARNI

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Summer Semester, 2007

The members of the Committee approve the Dissertation of Prasad A. Kulkarni defended on June 27th, 2007.

David B. Whalley
Professor Directing Dissertation

Steve Bellenot
Outside Committee Member

Gary Tyson
Committee Member

Xin Yuan
Committee Member

Kyle Gallivan
Committee Member

Theodore Baker
Committee Member

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

This dissertation is dedicated to aai, baba, sonu, and Prajna.

ACKNOWLEDGEMENTS

First and foremost, I thank my dissertation advisor, Dr. David Whalley, for encouraging me to pursue a Ph.D. I could not have done this work without his incredible technical advice, personal guidance, promptness, and understanding. I have learnt a lot from him over the past five years, and I will remain forever grateful. Not any less, I also thank Dr. Gary Tyson, who has essentially functioned as my co-advisor for the last three years. He has been a constant source of great ideas, support, and encouragement.

Since no level of thanking will be enough, I will only mention my parents, sister, and (my soon-to-be-wife) Prajna here. I can only hope they realize how much I depend on their presence in my life.

Finally, I also express gratitude to Bill Kreahling, Clint Whaley, Steve Hines, Wankang Zhao, and my other lab-mates for their suggestions, comments, and above all for making our office a fun place to work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The phase ordering problem has been a long-standing problem in compiler optimizations. Different orderings of applying optimization phases by a compiler can result in different code generated, with potentially significant performance differences for many applications or even functions within applications. At the same time it is universally acknowledged that a single ordering of optimization phases will not produce the best code for all functions. The strict performance constraints in applications for embedded and high performance systems is making it increasingly important to quickly find the optimal or near-optimal order of applying optimization phases so that efficient code can be produced.

Given the huge search space of all possible orderings of optimization phases, finding the optimal phase ordering for each function was generally considered intractable. Furthermore, heuristic approaches to address the phase ordering problem frequently take too long to converge on a good solution. In this dissertation I will first describe two complimentary approaches to achieve faster searches for effective optimization sequences when using a genetic algorithm. We then further leverage our observation of huge redundancies in the typical phase order space to make exhaustive evaluation of the entire phase order space practical in most cases. We also analyze various properties of the optimization phase order space, and show how such analysis can be used to generate faster conventional compilers, and enhance commonly employed heuristic approaches to produce better solutions faster.

CHAPTER 1

Introduction

This chapter will lay the foundation for this dissertation by describing our research problem in the wider context of compilers and compiler optimizations. I will also attempt to delineate the impact our research work can potentially have on the field of compiler optimizations.

1.1 Role of Compilers and Compiler Optimizations

From the current perspective, a *compiler* in it's simplest form can be described as a software program used to transform some other *high-level* language program into *low-level* assembly program or object code, for execution on a specific computer system. Grace Hopper, working on the A-0 programming language for the UNIVAC at Remington Rand Inc., is generally accredited for leading the effort to develop the first widely used compiler for the same language in 1952 [1]. However, most of the early compilers, or *automatic programming* systems as they were then called, were mainly employed to provide a richer instruction set than the one provided by the native machine. Such a *synthetic* instruction set contained enhanced functionality such as floating-point instructions, index registers, and improved input-output commands lacking on the native computer [2]. Early compilers typically generated programs that were significantly slower than hand-written assembly programs, and this high cost prevented the wide scale adoption of such program transformation systems.

The FORTRAN-I compiler, released in 1957, was the first compilation system that seriously regarded efficient program generation as one of its main design goals [3, 2]. John Backus, the lead developer of the FORTRAN programming language and compiler, believed that the failure of compilers to generate efficient programs would critically hamper the wide scale use of high-level programming languages like FORTRAN. The successive years have witnessed a large investment of time and resources by compiler developers to discover and

implement innovative *optimization* techniques to improve the quality of the code generated by the compiler. This focus on code quality has resulted in the development of numerous compiler optimization techniques, and in the production of high quality code by many compilers.

1.2 The Phase Ordering Problem

As more and more compiler optimizations started being implemented in various compilers, it was soon realized that many optimization phases interact with each other. Each optimization phase attempts to apply a series of *transformations* consisting of a sequence of changes that try to improve program efficiency, while preserving the semantic behavior of the program. Many of these optimization phases use and share resources (such as machine registers), and also need specific conditions in the code to be applicable. As a result optimization phases interact with each other by enabling and disabling opportunities for other phases to be applied. Such interactions between optimization phases have been widely studied in the context of different compilers (with different sets of optimization phases) and different architectures [4, 5, 6, 7, 8]. Based on such studies it is now definitively known that phase interaction often causes different orders of applying optimizations phases to generate different code, with potentially significant performance variation among them. Therefore, finding the best order of applying optimization phases is important in application areas where performance is paramount, such as in high performance and embedded domains, so that more efficient code can be generated for each application. This challenge is commonly known as the *phase ordering problem* in compilers. Over four decades of research on the phase ordering problem has shown that the problem is challenging since a single order of optimization phases will not produce optimal code for every application [9, 10, 11, 12, 13, 7]. The best order depends on the program being optimized, the manner in which the optimization phases are implemented in the compiler, and the characteristics of the target machine.

1.3 Addressing the Phase Ordering Problem

Over the years there have been several attempts to reasonably address the phase ordering problem in optimizing compilers. A comprehensive survey of some of the more important work in this area will be provided in Ch 2. In this section I will briefly summarize the

general approaches investigated by researchers, before outlining our method to address the phase ordering problem.

A fundamental requirement for finding the optimal phase ordering is the ability to reorder optimization phases in the compiler. Most early compilers did not recognize phase ordering to be an important issue, and thus several compilers were initially designed without taking this problem into consideration. As a result different optimization phases expected their input in different formats, commonly the output format of some earlier phase, which in effect serialized and fixed the order of optimization phases in that compiler. The growing awareness of the potential benefits of adapting the phase ordering led to the development of many compilers that enforced all optimization phases to operate on a single intermediate language representation of the program [14, 4]. Consequently, each optimization phase can be applied repetitively and in any order, which neutralized many simple phase ordering issues. Another approach adopted by some researchers is to combine two or more phases that are very sensitive to phase ordering into a single optimization phase [15, 16].

The successive years saw an exponential growth in computing power, which made possible, and gave rise to the field of *iterative compilation* [7]. Iterative compilation is the method of iteratively compiling and evaluating an application with sequences of different optimization-parameter pair values in an attempt to achieve the best performance. A naive solution to the phase ordering problem is to exhaustively evaluate the performance of all possible orderings of optimization phases. However, this approach requires the resolution of two sub-problems, both of which have always been considered infeasible for production quality compilers. The first sub-problem is to exhaustively *enumerate* all possible orderings of optimization phases. This enumeration is impractical since the phase ordering space has a tendency to quickly become infeasible to completely explore in the face of several different optimization phases that are typically present in current compilers, with few restrictions on the ordering of these phases. The second sub-problem is to *evaluate* the performance of all the enumerated orderings to find the optimal performance. Performance evaluation requires execution of the application, which is typically much more expensive than simply compiling the code. Furthermore, many embedded development environments do not support compilation, and hence require simulation instead of native execution, which is often orders of magnitude more expensive. These requirements were challenging enough to deter any previous attempts at exhaustive phase order space evaluation over all the optimization

phases in a mature compiler. Most research either used heuristic algorithms to evaluate only a portion of the optimization space [11], or performed exhaustive evaluation over a small subset of the available optimization phases to study optimization space characteristics [6].

During our research, we successfully demonstrated that although the complete optimization phase order space is extremely large, it contains a significant amount of redundancy, such that many different orderings of optimization phases ultimately produce the same code. Thus, by using various pruning techniques it is often possible to exhaustively evaluate all possible ways that code can be generated for the optimization phases in our compiler, and determine the *optimal* performing phase orderings with a very high degree of accuracy for most of the functions in our benchmark suite. In this dissertation, I will describe the techniques we used to prune the phase order search space to generate all possible *distinct function instances* that can be produced by changing the optimization phase orderings in our compiler, for the vast majority of the functions that we studied. I will also explain our approach of using simple and fast estimation techniques to reduce the number of program evaluations to quickly determine the optimal, or near-optimal, performing function instance. By using different correlation techniques, we show that our method of performance estimation is highly accurate for our purposes. Finally, exhaustive evaluation of the phase order space over a large number of functions provided us with a large data set, which we have analyzed to determine various properties of the optimization phase order space. I will explain how we have used this information to enhance our default *batch* compiler, as well as to suggest improvements to earlier heuristic methods to address the phase ordering problem.

1.4 Contributions and Potential Impact of Our Research

Our research is the first to demonstrate that for a large majority of the functions it is possible to exhaustively evaluate the optimization phase order space in a reasonable amount of time to find the phase ordering resulting in (near) optimal code being produced by the compiler. This single result can potentially have a significant impact on further research in addressing/solving the phase ordering problem in optimizing compilers by challenging the acceptance of the infeasibility of exhaustive phase order space evaluation. At the same time, the methodologies and algorithms we used, and the results we obtained can also be used in various other research problems in computer science and beyond.

The major contributions of our research can be concisely stated as follows:

1. We demonstrated how domain knowledge specific to sequences of optimization phases can be used to detect huge redundancies in the phase order space [17, 18]. We developed techniques to detect this redundancy to speed up already existing heuristic phase order search algorithms.

2. Instead of explicitly enumerating all possible sequences of optimization phases (which is indeed impractical), we rephrased the phase ordering problem to take advantage of redundant sequences, and made the problem more manageable [8]. We then developed new algorithms to make it possible to exhaustively enumerate the phase order space in a reasonable amount of time for most of the functions in our benchmark suite [19].

3. We simplified the program performance estimation problem, and justified the use of cheaper estimation techniques to measure program performance by employing well-known statistical methods to demonstrate their correlation with more expensive program simulations [20].

4. We developed techniques to gather relevant information about the phase order space to gain a better understanding of this space, and used this information to improve batch compilation [8], as well as to compare and evaluate different heuristic search algorithms, and proposed new algorithms that exploit the available information to improve these searches [21].

1.5 Outline of My Dissertation

This section will describe the layout of the remainder of this dissertation. Chapter 2 describes the related work of searching for effective ways to apply compiler optimizations. Chapter 3 presents the experimental framework we used for our research of the phase ordering problem.

It is worthwhile to point out here that my dissertation research is a natural progression of the work we did as part of my Master's thesis [22]. In my thesis, we presented the performance improvements over our default *batch* compilation that can be achieved by finding function-specific optimization phase orderings. A genetic algorithm based approach was used to find distinct phase orderings for every function. During that work we had observed significant redundancy in the successive orderings found by the genetic algorithm. It was quite

obvious that finding techniques to eliminate this redundancy has the potential of making the searches for effective optimization phase orderings faster and more productive. Chapter 4, will describe our efforts to improve efficiency when using genetic algorithm searches, by developing techniques to eliminate redundant sequences, and find better successor sequences.

During our genetic algorithm experiments, our techniques were able to avoid over 87% of program executions. This result indicated the presence of significant redundancy in the optimization phase order space. This observation also encouraged us to enhance our techniques of exploiting redundant phase orderings further to pursue the ultimate goal of finding the *optimal* phase ordering by performing an exhaustive search. However, even the task of enumerating all possible permutations of optimization phases is acknowledged to be impractical. Chapter 5 describes the approach we used to enumerate all distinct *function instances* that can be produced by all possible phase orderings, without explicitly listing each possible phase ordering permutation.

The next task to find the optimal phase ordering is to evaluate the performance of each distinct function instance (for every function) by execution or simulation of the application. Executing the application to determine its performance is generally orders of magnitude more expensive than just generating the code. Simulation, which is required in many embedded application development environments is even more time-consuming. Chapter 5 describes our approach, and the techniques we used to quickly find the optimal phase ordering with a very high degree of probability.

Exhaustively evaluating the optimization phase order space over a very large suite of functions provided us with a very large data set containing imformation about phase interactions, and other optimization phase properties. We have attempted to employ some of this information to improve conventional compilation. We also used our knowledge of the entire phase ordering space to compare commonly used heuristic algorithms that search for different phase orderings. We suggest improvements for current algorithms and suggest new algorithms as well. These experiments and evaluations are presented in Chapter 6. Some future directions for this research are outlined in Chapter 7, and finally our conclusions are presented in Chapter 8.

CHAPTER 2

Related Work

Modern compilers make extensive use of optimizations to improve program performance. Many compiler optimizations are highly dependent on both the program characteristics as well as the architectural environment on which the program is run. Architectural features like the number of registers, cache size, pipeline depth etc. can have a huge impact on the performance delivered by such optimizations. A compiler usually performs some form of static program analysis based on simplified machine models. However, the machine models used are inherently inaccurate, and cannot account for the behavior of the entire system. At the same time, transformations are not independent in their effect on performance of the generated code making the compiler's task of selecting the best sequence of transformations difficult. Moreover, compilers typically use heuristics that are based on averaging observed behavior for a small set of benchmarks. This generality leads to loss of performance in most applications.

All the compiler optimization issues discussed above deal with only the selection, arrangement, and parameterization of optimization phases. As such, research on the strategies to apply compiler optimizations has been divided into three distinct groups: (1) phase parameterization, (2) phase selection, and (3) phase ordering. An approach termed as *iterative compilation* is used in many compilers to aggressively optimize programs in order to overcome these issues. In this approach, many different combinations of optimization/parameter pairs are applied to a program and their worth is determined by actually executing the resulting code and measuring the execution time. Such an approach does not suffer from undecidability issues and, given sufficient time will find the optimal program for the set of optimizations being performed. The obvious drawback is that compilation time dramatically increases. As a result, researchers have also studied static

techniques to pre-select the optimization strategy. In several approaches heuristic models are used to impose a limit on the number of cases studied during iterative compilation. In this chapter I will describe the existing research work that attempts to address these compiler optimization issues.

2.1 Phase Parameterization

Many compiler optimizations are parameterized. If a compiler contains m optimization phases, with n_i legal parameter values for each phase i ($1 \leq i \leq m$), then the phase parameterization space is: $(n_1 * n_2 * ..... * n_m)$. Thus the space parameterization space can be very large. These parameters are either fixed, or commonly selected by incomplete, and simplified machine models, and as a result are frequently sub-optimal. For example, the number of times a loop is duplicated when applying loop unrolling is called the unroll factor. High unroll factors for loop unrolling can improve performance by reducing the loop overhead and providing for better instruction scheduling. However, high unroll factors also increase the possibility of cache overflow, and depending on machine cache sizes can degrade performance. Similarly, loop tiling is an optimization that typically improves memory heirarchy performance by increasing the temporal data locality. However, this optimziation can also hurt performance if it is incorrectly parameterized, and needs to be carefully tuned to obtain maximum benefit.

The small area of the transformation space formed by applying loop unrolling (with unroll factors from 1 to 20) and loop tiling (with tile sizes from 1 to 100) was analyzed for a set of three program kernels across three separate platforms [23, 7]. The study showed that the optimization space is highly non-linear, and the best parameters depend largely on the application and the processor. Their iterative strategy using a grid-based search algorithm was able to find good transformation settings by examining a very small fraction of the optimization space. One important deduction from this work was that typical program search spaces generally contain many local minima, and so heuristic algorithms need to be robust enough to not get trapped in a local minimum. In some related research the performance of many different heuristic and machine learning algorithms that search for effective phase parameters was evaluated [24]. The general observation was that different search algorithms do not differ much in their efficiency. Researchers have also incorporated

static cache models during iterative compilation, and have shown that cache models can reduce the number of executions by 50% [25]. This research also reported that searching for the best parameters using only static models yields performance improvements that are far less than those obtained by iterative compilation.

The ATLAS project [26] used iterative compilation to generate dense numerical linear algebra library called the BLAS. They call their approach *automated empirical optimization of software* (AEOS). The ATLAS approach works best for the level-3 BLAS, which performs matrix-matrix multiplications. ATLAS runs in two phases. In the first phase, ATLAS runs micro-benchmarks to approximately determine the values of some architectural features, such as L1 cache size, number of floating-point registers, and the presence of multipy-add instruction in the specific machine's instruction set. In the next phase, an extensive empirical search is performed to find the most effective parameters for cache and register level tiling, loop unrolling, and scheduling. The machine parameters estimated in the first step are used to limit the empirical search in the next phase.

The PHiPAC [27] project used iterative techniques to optimize linear algebra routines. The FFTW [28, 29, 30] project used AEOS-like techniques to optimize *fast fourier transforms*. Other projects with AEOS-like designs include *Optimqr*, for creating solvers for sparse systems of linear equations [31], software for designing signal processing algorithms [32], and *Tune*, for designing models and transformations for memory-friendly programming [33].

Many researchers have also employed static models at compile time to search for good optimization parameters. This includes cache models to select tile sizes [34], and to compute loop unroll factors [35]. Research on phase parameterization is, however, only concerned with selecting the correct parameters for a few compiler optimizations in isolation, and does not consider all compiler optimizations. In constrast, my dissertation research, instead, considers the order in which all optimization phases should be applied.

2.2 Phase Selection Problem

Most current mainstream compilers implement many different optimization phases. As such, not all the phases may be able to achieve a performance improvement for all input functions. For example, loop unrolling usually improves performance. However, depending on machine characteristics like size of the instruction cache and number of allocatable registers, loop unrolling can actually degrade performance by causing cache or register overflow. Many

compilers provide command-line flags that can be turned on/off to select the phases to be applied to each function or program [36]. The *phase selection problem* is the problem of selecting the ideal compiler flags to achieve the best program performance for each application. If a compiler provides n optimization flags, then there are at most 2^n possible selections of compiler optimization options.

The search space of all possible combinations of optimization flags can be very large. Chow and Wu applied a technique called *fractional factorial design* [37] to systematically design a series of experiments to select a good set of program-specific optimization flags [38]. Each set of experiments determines some effects and interactions between compiler switches. A relatively few number of runs were needed to detect most interactions. However, much of the work needed to select the experiments, and isolate higher-level optimization effects was manually performed in this research.

Some researchers have also used *orthogonal arrays* to create a fractional factorial design of experiments to partially explore the space of all possible combinations of compiler switches [39]. This research used orthogonal arrays of strength two to compute the effect each compiler option will have on the final application performance. Such statistical analysis allowed them to turn on/off certain compiler flags. Follow-up work on the same topic allowed the researchers to generate a compiler setting that is optimized for a collection of applications [40]. In addition to looking at the performance effect of each option, this research also considered the interaction between pairs of optimizations in the presence of other options being turned on.

Granston and Holler propose a tool, called *Dr. Options*, for automatic recommendation of compiler options [41]. This tool uses a set of deterministic rules for when to turn on certain options based on interviewing compiler writers, their experience, and results from literature. The tool gathers static information about the current function, such as characteristics of loops and data access patterns, and optionally also uses user-supplied, and profiling information to guide the compiler options selection process.

Pan and Eigenmann developed three different algorithms to quickly select good compiler settings [42]. The first algorithm, called *batch elimination*, requires a single experimental run per optimization phase to identify and turn off that phase if it causes a performance degradation. This algorithm does not consider interactions between optimizations. *Iterative elimination* is the second algorithm that takes n experimental runs (n is the number of

phases in the compiler) per iteration to switch off one flag with the most negative effect. The algorithm continues until an iteration is reached when no optimization is causing a performance degradation. The third algorithm, called *combined selection*, combines the previous two approaches. During each iteration, after identifying the phases with negative effects, this algorithm tries to eliminate all these negative optimizations one by one in a greedy fashion.

Research on the problem of phase selection only attempts to find the best set of compiler options to apply, and does not vary the ordering of the phases. Different from this, my goal is to find the ideal ordering of compiler optimizations. This phase ordering problem in fact subsumes the phase selection problem.

2.3 Phase Ordering

Optimization phase ordering is a long standing problem in compilers and as such there is a large body of existing research on this topic. An interesting research study investigating the decidability of the phase ordering problem in optimizing compilation proved that finding the optimal phase ordering is *undecidable* in the general schemes of iterative compilation and library generation/optimization [43]. However, their hypothesis assumes that the set of all possible programs generated by distinct phase orderings is infinite. This hypothesis is rational since optimizations such as loop unrolling, and strip mining can be applied an arbitrary number of times, and can generate as many distinct programs. In practice, however, compilers typically impose a restriction on the number of times such phases can be repeated in a normal optimization sequence. Thus, finding the optimal phase ordering is decidable in most current compilers, albeit very hard.

Exhaustive evaluation of the entire optimization phase order space in a mature compiler has generally been considered infeasible, and, to the best of our knowledge, had never before been successfully attempted before my present work. Enumerations of search spaces over a small subset of available optimizations have, however, been attempted. One group exhaustively enumerated a 10-of-5 subspace (optimization sequences of length 10 from 5 distinct optimizations) for some small programs [6]. Each of these enumerations typically required several processor months even for small programs. The researchers found the search spaces to be neither smooth nor convex, as well as noticed the difficulty in predicting the best optimizations in most cases.

Since exhaustive phase order search space exploration had generally been conceived as infeasible, researchers investigated the problem of finding an effective optimization phase sequence by aggressive pruning and/or evaluation of only a portion of the search space. This area has seen the application of common *artificial intelligence* search techniques to search the optimization space. Hill climbers, simulated annealing, as well as genetic algorithms have been used during iterative searches to find optimization phase sequences better than the default one used in their compilers [11, 12, 6]. Most of the results report good performance improvements over a fixed compiler sequence.

There have been several attempts to reduce the cost of iterative search to address the phase ordering problem. Agakov et al. used predictive modeling to automatically focus the search on those areas of the space likely to result in maximum performance improvement [44]. This approach uses program features to correlate the program to be optimized with previous knowledge in order to focus the search. In this approach a training set of programs is iteratively evaluated, and the shape of the spaces and program features are modeled. When a new program is encountered, program features are used to select the appropriate model, which then biases the search to a certain area of the space. Thus, the time for iterative search is significantly reduced.

Model driven or analytical approaches to address the phase ordering problem have also been attempted. Such approaches attempt 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. Such approaches have minimal overhead since additional profile runs of the application are generally not required.

Whitfield and Soffa developed a common framework based on a formal axiomatic specification language for describing the actions of optimizations and the conditions for performing them [10, 5]. This framework was employed to theoretically list the *enabling* and *disabling* interactions between optimizations, which were then used to derive an application order for the optimizations. It was found that many pairs of optimization phases may not enable or disable another, while in many of the remaining cases the interaction relationship is theoretically one-way. For such phases it is trivial to find the best order of applying optimization phases. The main drawback of this work was that in cases of cyclic interactions between optimization phases it was not possible to automatically determine a good ordering without detailed information about the compiler and application.

Follow-up work on the same topic has seen the use of additional analytical models, including code models, resource models (such as for cache, registers, and computation), and optimization models, to determine and predict other properties of optimization phases such as the *impact* of optimizations [45], and the *profitability* of optimizations [46]. 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.

The Unified Transformation Framework (UTF) was proposed to provide a uniform and systematic representation of iteration reordering transformations (implemented for loop interchange, reversal, skewing, distribution, fusion, alignment, interleaving, tiling, coalescing, scaling, statement reordering, and index set splitting) and their arbitrary combinations [47]. It is possible using UTF to represent a sequence of iteration reordering transformations as a sequence of parameters. UTF allows a compiler to transform the optimization phase order space into a polyhedral space, which is considered by some researchers to be more convenient for a systematic exploration than the original space [48]. Researchers have further proposed a random heuristic search algorithm independent of architecture, language, and environment to locate high-performance points in the space of iteration reordering transformations [48]. However, this work is only restricted to loop optimizations, and needs to be extended to other optimizations before it's merit can be evaluated for typical iterative compilers.

In contrast to all earlier approaches to address phase ordering, my dissertation work is the first and only successful attempt to exhaustively evaluate the phase order space over all the phases in a compiler to find the optimal phase ordering for each function.

2.4 Other Iterative Compilation Approaches

In additional to the above work on phase selection, ordering, and parameterization there has been some other work as well that used empirical optimization techniques. A method called Optimization-Space Exploration [13] combines phase selection with the search for good optimization parameter settings. This algorithm uses a compile-time pruning technique, and uses compiler writer's knowledge to select a small set of optimization-parameter settings that are known to perform well. At run-time, this algorithm iteratively attempts to find better optimization parameters by merging the beneficial ones in the previous iterations. Static performance estimators are used to reduce the search time. This approach is very general

since code for critical segments are actually generated and a static performance estimation is applied. Thus, any set of optimizations can be used in this approach.

There has been some research investigating the performance of exhaustive searches in relatively small search spaces to find optimization sequences to improve code for individual benchmarks. Exhaustive techniques have been developed to search for optimal instruction sequences [49] or to eliminate branches [50]. However, these approaches can only be used in very limited contexts.

2.5 Static Performance Estimation

Several prior studies have used static performance estimations to avoid program executions during iterative compilation [51, 52, 13]. Encouragement for replacing program executions with faster static estimations was provided by earlier studies, such as the one by Wagner et al., who presented a number of static performance estimation techniques to determine the relative execution frequency of program regions, and measured their accuracy by comparing them to profiling [53]. They reported that in most cases static estimators provided sufficient accuracy for their tasks. Researchers have experimented with detailed cache models [51], along with a model for predicting branches [13] to reduce the number of program executions needed during iterative compilation.

Other researchers have also varied how optimizations are applied and have instead used static estimations of performance to reduce the search time. One method searches through the different ways to apply loop fission, fusion, interchange, and outer loop unrolling in an attempt to optimize loop nests [54]. This method does not actually generate code, but instead uses an estimate based on the original loop nest and the potential benefit for a transformation. Thus, their approach is limited since the estimator only works on the set of optimizations being considered. The search space is pruned in different ways. The decisions regarding how to apply some optimizations, such as outer loop unrolling factors, are made independently from other optimizations since it was felt that it would not be affected by how inner loops would be optimized. The number of loops to be varied when tuning other optimizations, such as tile size and loop interchange, are also limited. Other methods for integrating different optimization phases were also studied [55, 56].

The method of static performance estimation I used during my research is most similar to the approach of *virtual execution* used by Cooper et al. [52] in their ACME system of

compilation. In the ACME system, Cooper et al. strived to execute the application only once (for the un-optimized code) and then based on the execution counts of the basic blocks in that function instance, and careful analysis of transformations applied by their compiler, determine the dynamic instruction counts for other events, such as function instances. With this approach, ACME has to maintain detailed state, which introduces some amount of additional complexity in the compiler. In spite of detailed analysis, in a few cases ACME is not able to accurately determine the dynamic instruction count due to the types of optimizations been applied, occasionally resulting in small errors in their computation.

2.6 Evaluating Heuristic Search Algorithms

In spite of the wide-spread use of heuristic approaches, there have been few attempts to evaluate and compare their properties and performance in the context of the characteristics of the phase order space. Kisuki et al. analyzed the performance of five different search algorithms, and reported the observation that heuristic algorithms do not differ much in their efficiency [24]. However, this study was performed on a space of only two optimizations (with different parameters), and did not take into account properties of the phase order space. A more detailed evaluation was performed by Almagor et al. [6], which attempted to relate features of the phase order space with the efficiency and performance of different heuristic algorithms. This study was, however, incomplete since they had restricted their sequence length to be of a fixed size. This earlier work also did not have access to the entire phase order space features, and lacked a knowledge of the optimal phase order performance.

CHAPTER 3

Experimental Framework Common throughout the Dissertation

We selected the Very Portable Optimizer (VPO) [57] as our compiler backend to conduct experiments and verify our hypothesis. VPO was a part of the DARPA and NSF co-sponsored National Compiler Infrastructure project to produce and distribute a compiler infrastructure to be used by compiler researchers in universities, government, and industry. 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. I have already discussed that repetitively applying optimization phases is considered an effective way to address simple inefficiencies due to phase ordering [4]. VPO compiles and optimizes one function at a time. This is important since different functions may have very different best orderings, so any strategy that requires all functions in a file to have the same phase order will almost certainly not be optimal. At the same time, restricting the phase ordering problem to a single function helps to make the phase order space more manageable. VPO has been targeted to produce code for a variety of different architectures. Some of our initial experiments were conducted on an Ultra SPARC III processor. The SPARC provided a high performance platform along with support for native compilation and execution. Due to our focus on embedded applications, we ultimately moved over to the StrongARM SA-100 platform, using Linux as its operating system. VPO was used as a cross-compiler executing on fast x86 machines while generating code for the ARM. We used the SimpleScalar set of functional and cycle-accurate simulators [58] for the ARM to get dynamic performance measures.

Table 3.1 describes each of the 15 candidate code-improving phases that we used during

our exhaustive and heuristic explorations of the optimization phase order search space. In addition, *register assignment*, which is a compulsory phase that assigns pseudo registers to hardware registers, must be performed. In our experiments VPO implicitly performs register assignment before the first code-improving phase in a sequence that requires it. Two other optimizations, *merge basic blocks* and *eliminate empty blocks*, were removed from the optimization list used for our searches since these optimizations only change the internal control-flow representation as seen by the compiler and do not directly affect the final generated code. These optimizations are now implicitly performed after any transformation that has the potential of enabling them. After applying the last code-improving phase in a sequence, VPO performs another compulsory phase that inserts instructions at the entry and exit of the function to manage the activation record on the run-time stack. Finally, the compiler also performs predication and instruction scheduling before the final assembly code is produced. These last two optimizations should only be performed late in VPO's compilation process, and so it was not possible to include these optimizations in our regular set of phases for searching the phase order space.

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 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. 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*. These phases can be performed in any order after *register allocation* is applied. *Register allocation* itself can only be performed after *instruction selection* so that candidate load and store instructions can contain the addresses of arguments or local scalars. There are a set of phases that require the allocation of hardware registers and must be performed after *register assignment*. These phases include *register allocation*, *common subexpression elimination*, *dead assignment elimination*, and most loop transformations, including *loop unrolling*, *loop-invariant code motion*, *loop strength reduction*, *recurrence elimination*, and *induction variable elimination*. As mentioned earlier, *register assignment* is automatically performed before the first phase that requires it. Finally, we restrict *loop unrolling* to be active at most once in each sequence.

In this study we were only investigating the phase ordering problem and did not vary

Table 3.1: Candidate Optimization Phases Along with their Designations

Optimization Phase	Gene	Description
branch chaining	b	Replaces a branch or jump target with the target of the last jump in the jump chain.
common subexpression elimination	c	Performs global analysis to eliminate fully redundant calculations, which also includes global constant and copy propagation.
remove unreachable code	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 runtime 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.
minimize loop jumps	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.
reverse branches	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 together where the instructions are linked by set/use dependencies. Also performs constant folding and checks if the resulting effect is a legal instruction before committing to the transformation.
remove useless jumps	u	Removes jumps and branches whose target is the following positional block.

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. Note that VPO is a compiler backend. 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 most of this research, 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.

Table 3.2: MiBench Benchmarks Used in the Experiments

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

Note that some phases in VPO represent multiple optimizations in many compilers. However, there exist compilers, such as GCC, that have a greater number of distinct optimization phases. Unlike VPO, most compilers are more restrictive regarding the order in which optimizations phases are performed. In addition, the more obscure a phase is, the less likely that it will be successfully applied and affect the search space. While one can always claim that additional phases can be added to a compiler or that some phases can be applied with different parameters (e.g., different unroll factors for loop unrolling), completely enumerating the optimization phase order space for the number of phases applied in our compiler had never before been accomplished to the best of our knowledge.

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 [59]. We selected two benchmarks from each of the six categories of applications present in MiBench. Table 3.2 contains descriptions of these programs. VPO compiles and optimizes individual functions at a time. The 12 benchmarks selected contained a total of 244 functions, out of which 88 were executed with the input data provided with each benchmark. Detailed information about the functions in each benchmark is provided in Appendix A.

CHAPTER 4

Techniques for Faster Genetic Algorithm Searches

Researchers attempting to address the phase ordering problem in optimizing compilers have always had the ultimate goal of finding the *optimal* phase ordering for each application or function. However, this ultimate goal has always been considered intractable for production quality compilers, and over all the optimization phases present in their compilers. There are two main contributing reasons for this prevalent notion of infeasibility.

First, the space of all possible orderings of optimization phases is extremely huge. Current compilers contain numerous different optimization phases, with few restrictions on the ordering of different phases. For example, VPO contains 15 different re-orderable phases. At the same time, phases can enable each other by creating new opportunities for application of other optimization phases. Thus, each phase can be active multiple times in the same phase sequence, which makes it impossible to statically determine the scope of the phase order space for each function. For example, with VPO the space can potentially be much greater than the 15! number of all possible permutations of optimization phases.

Second, evaluating the performance of each phase ordering requires a pass to compile, assemble, and link the application, in addition to executing the application, and verifying the generated output for correctness. Many embedded development environments do not natively support a run-time system, which implies that simulation (rather than execution) will be required to determine the phase ordering performance. Simulation can be orders of magnitude more expensive than native execution. Performing these operations to measure the relative performance of all possible phase orderings quickly becomes prohibitively expensive.

Moreover, based on phase interaction behavior, it is recognized that no single ordering of phases will be optimal for all applications. This means that it is not enough to only

perform the optimal phase ordering measurements once, but that since the best orderings can change, the measurements need to be repeated for each new application on every computer platform. Researchers have also attempted to study and classify the behavior and interactions between optimization phases. But, the phase interactions still remain ill-understood and unpredictable. Thus, it is not at all surprising that determining the optimal phase ordering was always considered infeasible, and there were few attempts in that direction. Instead, researchers attempted to use heuristic algorithms to probe only a part of the phase order space to find *effective* function-specific optimization phase orderings.

Heuristic algorithms, such as genetic algorithm and hill climbing, are commonly used to search for effective function or program-specific optimization phase orderings. While such searches have been shown to produce more efficient code than that produced by a single phase ordering, these approaches can be extremely slow because the application needs to be compiled and possibly executed to evaluate the effectiveness of each sequence. As part of my Masters's *thesis*, I studied the cost and performance impact of using genetic algorithms to find function-specific optimization phase sequences [22]. We found that even for small embedded applications the search process often required several hours to converge on a good sequence. Focusing on the same goal of improving searches for function-specific optimization phase sequences as part of my dissertation, we proposed and evaluated two complimentary approaches for achieving faster searches for effective optimization sequences when using a genetic algorithm. In this chapter, I will describe the techniques that we used to obtain dramatic improvements in our genetic algorithm searches, and what the observations from these experiments meant for my subsequent research in this field. These experiments did not use *loop unrolling* and *code abstraction* as available reorderable phases. Additionally, two phases, *merge basic blocks* and *eliminate empty blocks*, that were later removed from our list of reorderable phases, were included during the experiments described in this chapter. The experiments in this chapter were conducted on an Ultra SPARC III processor, unless indicated otherwise.

4.1 Background

Before describing the techniques to achieve faster genetic algorithm searches, I will first provide a brief background of some of my earlier research. This includes an introduction to the VISTA compilation framework, and an explanation of the genetic algorithm we employed

in our experiments.

4.1.1 Terminology

I will start this section by providing descriptions of some terminology that I use throughout my dissertation.

Active Phase : If the current function contains the enabling conditions for an optimization phase, then application of this phase will modify the function representation. A phase able to change the code generated for the function is called an *active phase*.

Dormant Phase : If an optimization phase is not able to find opportunities to modify the function representation, then this phase is called a *dormant phase*.

Function Instance : A *function instance* can be formally defined as any syntactically, semantically, and functionally identical representation of the source function. It is in essence a version of the function produced by some phase ordering sequence.

Batch Compiler : The VPO compiler operating in the default mode is called the *batch compiler*. Consequently, the phase ordering applied by the batch compiler is called the *batch sequence*. The default compiler aggressively applies optimizations in a fixed order, but repeats the sequence in a loop until some pass is attained during which no optimization is able to modify the function representation.

4.1.2 VISTA: VPO Interactive System for Tuning Applications

The experiments described in this chapter were conducted using VISTA. VISTA is a low-level interactive compilation framework [60, 61]. Figure 4.1 illustrates the flow of information in VISTA, which consists of a compiler and a viewer. The programmer initially provides a file to be compiled and then specifies requests through the viewer, which include sequences of optimization phases, manually specified transformations, and queries. The compiler performs the specified actions and sends program representation information back to the viewer. To evaluate the performance impact of a single optimization phase, or a sequence of phases, the compiler first applies all specified phases to the function under consideration. The compiler then optionally instruments the function using EASE [62], produces assembly code, links and executes the program, and gets performance measures from the execution. When the user

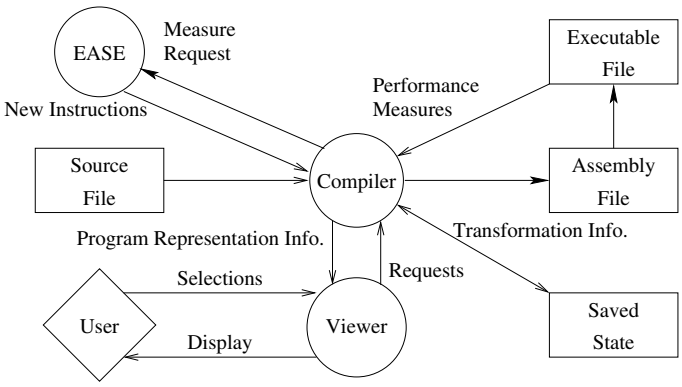


Figure 4.1: Interactive Code Improvement Process

chooses to terminate the session, VISTA writes the sequence of transformations to a file so they can be reapplied at a later time, enabling future updates to the program representation.

The interactiveness of VISTA is an important aspect of the environment. A user can view the program representation after each phase or transformation along with performance feedback to gauge the improvement during the tuning process. In addition, a user can manually specify transformations, which is particularly useful when there are architectural features that the compiler cannot exploit. Note that traditional compiler optimization phases can be applied even after manually specifying transformations.

Figure 4.2 shows a snapshot of the viewer with a history of a sequence of optimization phases displayed. Note that not only is the number of transformations associated with each optimization phase displayed, but also the improvements in instructions executed and code size are shown. Likewise, we can interface with a simulator to obtain improvements in cycle count and power consumption. This information allows a user to quickly gauge the progress that has been made in improving the function. The frequency of each basic block relative to the function is also shown in each block header line, which allows a user to identify the critical regions of a function.

VISTA allows a user to specify a set of distinct optimization phases and have the compiler attempt to find the best sequence for applying these phases for a given function. Figure 4.3 shows the different options that VISTA provides the user to control the search. The user

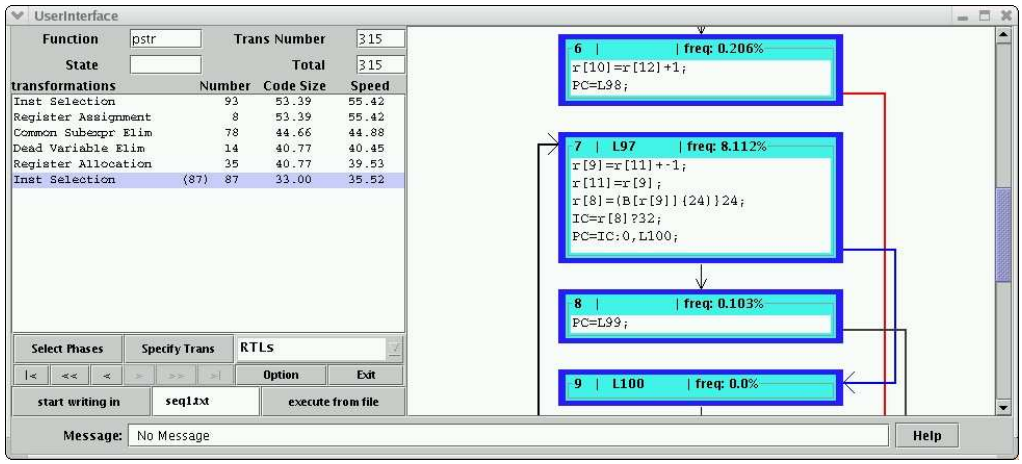


Figure 4.2: Main Window of VISTA Showing History of Optimization Phases

specifies the *sequence length*, which is the total number of phases applied in each sequence. Our experiments used the *biased sampling search*, which applies a genetic algorithm in an attempt to find the most effective sequence within a limited amount of time since in many cases the search space is too large to evaluate all possible sequences [63]. A population is the set of solutions (sequences) that are under consideration. The number of generations indicates how many sets of populations are to be evaluated. The population size and the number of generations must be specified, which limits the total number of sequences evaluated. These terms are described in more detail in the next section. VISTA also allows the user to choose dynamic and static weight factors, where the relative improvement of each is used to determine the overall fitness.

Performing these iterative searches is time consuming, typically requiring tens of minutes for a single function, and hours or days for an entire application even when using direct execution. Thus, VISTA provides a window showing the current search status. Figure 4.4 shows a snapshot of the status of the search selected in Figure 4.3. The percentage of sequences completed, the best sequence, and its effect on performance are given. The user can terminate the search at any point and accept the best sequence found so far.

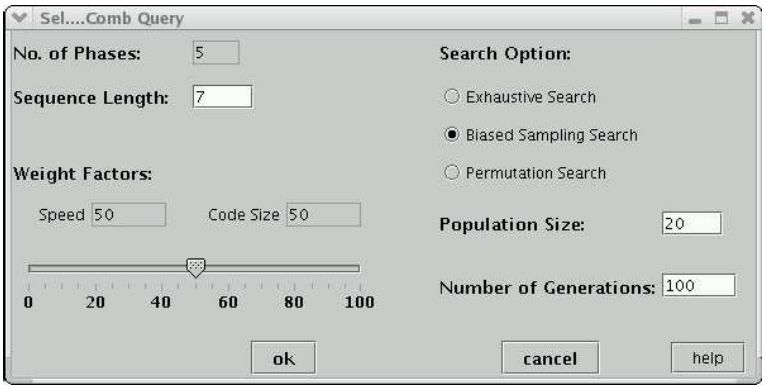


Figure 4.3: Selecting Options to Search for Possible Sequences

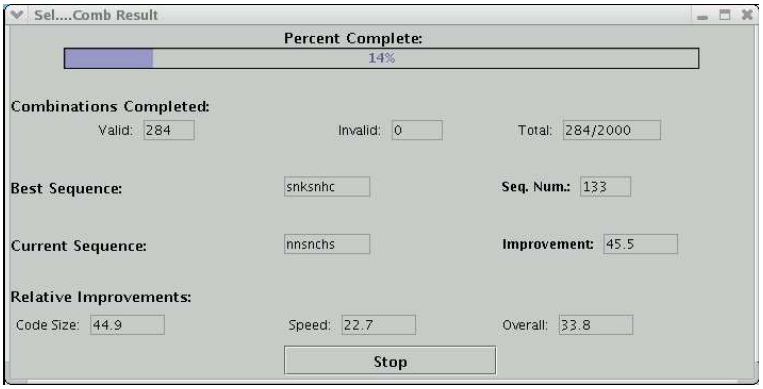


Figure 4.4: Window Showing the Search Status

4.1.3 Genetic Algorithm Performance Results

A genetic algorithm is a randomized biased sampling search technique that attempts to model the natural process of evolution in living beings [63, 64]. Past studies using genetic algorithms to generate better phase orderings have performed searches on entire applications [65, 11, 66]. In contrast, we performed our searches on individual functions, which require a greater number of compilations but results in better overall improvements [12]. In fact, most of the techniques we evaluated would be much less effective had we searched for a single sequence to be applied on an entire application. These experiments were conducted on 106 functions from six MiBench benchmarks, one from each available category (the first benchmark from

each category in Table 3.2).

We set the sequence (chromosome) length for the genetic algorithm to be 1.25 times the number of active phases that were applied for that function by the batch compiler. We felt this length was a reasonable limit that gives VISTA an opportunity to apply more active phases than what the batch compiler could accomplish. Note that this length is much less than the number of phases attempted during the batch compilation. The sequence lengths used in these experiments varied between 3 and 50 with an average of 14.15. We set the population size (fixed number of sequences or chromosomes) to 20 and each of our initial sequences was randomly initialized with candidate optimization phases. We performed 100 generations when searching for the best sequence for each function. We sorted the sequences in the population by a *fitness value* calculated using a 50% weight on speed and a 50% weight on code size. The speed factor we used was the number of instructions executed since this was a measure that could be consistently obtained, allowing us to obtain baseline measurements within a reasonable period of time. Similar speed factors have been used in earlier studies as well [11, 12]. It is possible to obtain a more accurate measure of speed by using a cycle-accurate simulator. However, the main point of our experiments was to evaluate the effectiveness of techniques for obtaining faster searches, which can be applied with any type of fitness evaluation criteria.

At each generation (time step) we removed the worst sequence and three others from the lower (poorer performing) half of the population chosen at random. Each of the removed sequences were replaced by randomly selecting a pair of the remaining sequences from the upper half of the population and performing a crossover (mating) operation to create a pair of new sequences. The crossover operation combined the lower half of one sequence with the upper half of the other sequence and vice versa to create two new sequences. Fifteen (75%) sequences are then candidates for being changed (mutated) by considering each optimization phase (gene) in the sequence. Mutation of each phase in a sequence occurred with a probability of 10% and 5% for the lower and upper halves of the population, respectively. When an optimization phase is mutated, it is randomly replaced with another phase. The four sequences subjected to crossover and the best performing sequence are not mutated. Finally, identical sequences in the same population were replaced with randomly generated sequences

Figures 4.5, 4.6, and 4.7 show the percentage improvement that we obtained for the

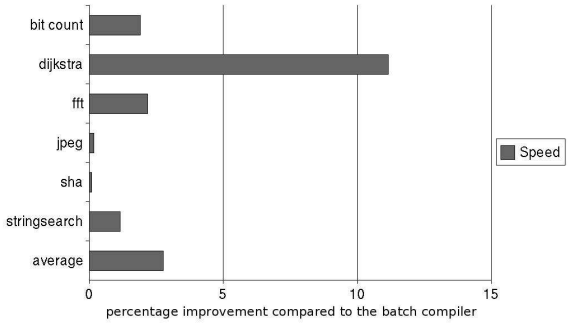


Figure 4.5: Speed Only Improvements for the SPARC

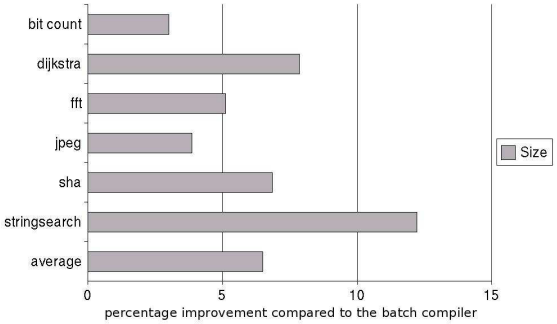


Figure 4.6: Size Only Improvements for the SPARC

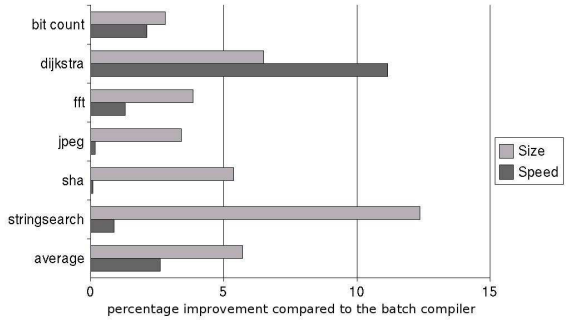


Figure 4.7: Size and Speed Improvements for the SPARC

SPARC when optimizing for speed only, size only, and 50% for each factor, respectively. The baseline measures were obtained using the batch VPO compiler, which iteratively applies optimization phases until no more improvements can be obtained. This baseline is much more aggressive than always using a fixed length sequence of phases [61], which makes our performance improvements look modest. More information about the VISTA compilation framework, or the specifics of our genetic algorithm searches can be obtained from previous publications [60, 12, 61].

4.2 Reducing the Search Overhead

Performing a search for an effective optimization phase sequence can be quite expensive, perhaps requiring hours or days for an entire application even when using direct execution instead of simulation to evaluate performance. Our main motivation for speeding up these searches was to make iterative compilation more attractive to use. Another benefit is that the heuristic searches can now be made more aggressive by increasing the number of generations, to potentially produce greater performance improvements. The following subsections describe the methods we used to reduce the search overhead, and the results of applying these methods.

4.2.1 Methods for Reducing the Search Overhead

Iterative compilation performs the following tasks to obtain dynamic performance measurements for a single sequence. (1) The compiler applies the optimization phases in the order specified by the sequence. (2) The generated code for the function is instrumented, if required, to obtain performance measurements. The assembly code for that function and the remaining assembly code for the functions in the current source file are written to a file. (3) The newly generated assembly file is assembled. (4) The object files comprising the entire program are linked together into an executable by a command supplied in a configuration file. (5) To obtain performance measurements, the program is executed using a command in a configuration file, which may involve direct execution or simulation. (6) The output of the execution is compared to the desired output to provide assurance that the new sequence did not cause the compiler to generate invalid code.[1] Tasks 2-6 often dominate the search time,

[1]It is possible that a new optimization sequence can cause the generated code to produce incorrect output. In the rare case when this happened, we assigned a poor fitness value to the sequence so that it will not be

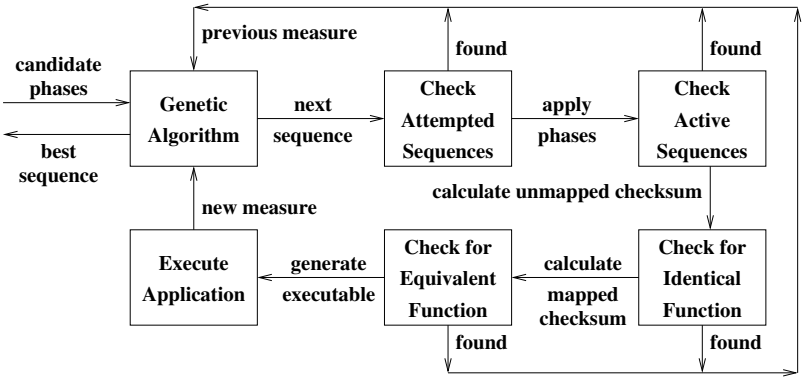


Figure 4.8: Methods for Reducing Search Overhead

which is probably due to these tasks requiring I/O and task 1 being performed in memory.

The following subsections describe methods to reduce the search overhead by eliminating redundant program compilations/evaluations. Figure 4.8 illustrates the order in which the different methods are attempted. Each optimization phase sequence generated by the genetic algorithm is checked by up to four methods. The methods are ordered according to cost. Each method handles a superset of the sequences handled by the methods applied before it, but the later methods are more expensive. The first method checks if the attempted sequence has been previously encountered for the function. If so, then the compilation by applying these phases is avoided. The second, third, and fourth methods are used to avoid the execution of the application, which comprise tasks 2-6 described earlier.

Finding Redundant Attempted Sequences

Sometimes the same optimization phase sequence is reattempted during the search. Consider Figure 4.9, where each optimization phase in a sequence is represented by a letter. The same sequence can be reattempted due to mutation not occurring on any of the phases in the sequence (e.g. sequence i remaining the same in Figure 4.9). Likewise, a crossover operation or mutation changing some individual phases can produce a previously attempted sequence (e.g. sequence k mutates to be the same as sequence j before mutation in Figure 4.9). A hash table of attempted sequences along with the performance result for each sequence is

selected by the genetic algorithm.

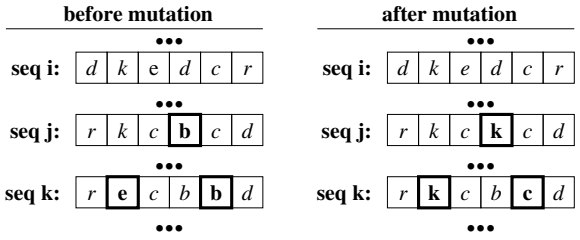


Figure 4.9: Example of Redundant Attempted Sequences

maintained. If a sequence is found to be previously attempted, then the evaluation of the sequence is not performed and the previous result is used. This technique of using a hash table to capture prior attempted solutions had been previously used in other research to reduce search time [11, 66, 12].

We realized that different sequences with the same attempted phases may generate the same code since some optimization phases are independent in that the order in which they are performed cannot affect the final code that is being generated. For instance, consider applying branch chaining before and after register allocation. Branch chaining does not change the live range of any variable that is a candidate for register allocation. Likewise, register allocation does not affect branch chaining since it does not affect conditional branches or unconditional jumps. Both branch chaining and register allocation will neither inhibit nor enable the other phase. Therefore, we statically identified for each optimization phase whether or not it is independent with each of the other phases. Rather than directly using the attempted sequence in the hash, we instead first sort the phases within the sequence so that two consecutively applied phases that are independent are always performed in the same order. We then used the sorted sequence of phases when accessing the hash table. Using the sorted sequence allowed more redundant sequences to be detected so more compilations were avoided. The x entries in Table 4.1 indicate which optimizations are independent of one another in the VPO compiler. For instance, branch chaining (b) is independent of register allocation (k).

We used our experience and insight in deriving the information in this table indicating which optimization phases are independent of one another. We inserted sanity checks when running our experiments to ensure that this information was correct. We were surprised

Table 4.1: Independent Optimization Phases

Optimization Phase	Gene	b	c	d	e	h	i	j	k	l	m	o	q	r	s	u
branch chaining	b	x			x				x	x		x	x		x	
comm subexpr elim	c		x				x							x		x
remv unreach code	d			x		x		x				x	x		x	x
remv useless blocks	e	x			x			x	x		x	x	x		x	
dead asg elim	h			x		x	x	x			x		x			x
block reordering	i		x			x	x		x	x		x	x		x	
min loop jumps	j			x	x	x		x				x	x		x	
register allocation	k	x			x		x		x					x		x
loop trans	l	x					x			x		x				x
merge basic blocks	m				x	x					x		x			
eval order determ	o	x		x	x		x	x		x		x		x		x
strength reduction	q	x		x	x	x	x	x			x		x	x		x
reverse branches	r		x						x			x	x	x	x	
instruction select	s	x		x	x		x	x						x	x	x
remv useless jumps	u		x	x		x			x	x		x	x		x	x

***x* indicates if the two phases are independent**

that our initial reasoning was often incorrect and we had to rectify our independence table on several occasions.

Finding Redundant Active Sequences

A dormant phase is unable to apply any transformations. As one would expect, only a subset of the attempted phases in a sequence will typically be active. It is common that a dormant phase may be mutated to another dormant phase, but it would not affect the compilation. Figure 4.10 illustrates how different attempted sequences can map to the same active sequence, where the bold boxes represent active phases and the non-bold boxes represent dormant phases. A second hash table is used to record sequences where only the active phases are represented. As when accessing the attempted hash table, we also sort the phases in the active sequence so that two consecutive independent phases are always applied in the same order.

Detecting Identical Code

Sometimes identical code can be generated from different active sequences. Often different optimization phases can be applied and can have the same effect. Consider the two different ways that the pair of instructions in Figure 4.11 can be merged together. Instruction selection

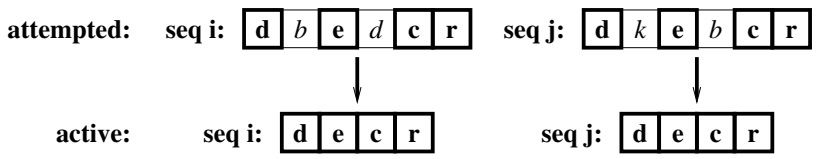


Figure 4.10: Example of a Redundant Active Sequence

original code segment
`r[2]=1;`
`r[3]=r[4]+r[2];`
after instruction selection
`r[3]=r[4]+1;`
	original code segment
	`r[2]=1;`
	`r[3]=r[4]+r[2];`
	after constant propagation
	`r[2]=1;`
	`r[3]=r[4]+1;`
	after dead assignment elimination
	`r[3]=r[4]+1;`

Figure 4.11: Different Optimizations Having the Same Effect

symbolically merges the instructions and checks to see if the resulting instruction is legal. The same effect in this case can be produced by constant propagation (actually part of common subexpression elimination in VPO) followed by dead assignment elimination.

We also found that while some optimization phases are not independent, the order in which they are applied often do not affect the generated code. For instance, branch chaining causes a transfer of control to go directly to the end of a chain of unconditional jumps. It is possible that one of those unconditional jumps in the chain can become unreachable code after performing branch chaining. However, this is unlikely to happen.

To lower the search overhead, it was important to efficiently detect when different active sequences generate identical code. A search may result in thousands of unique function instances, which may be too large to store in memory and be very expensive to access on disk. The key realization in addressing this issue was that while we wanted to detect all occurrences of identical function instances, we could tolerate occasionally treating different instances as being identical since the sequences within a population are sorted and the best sequence found by the genetic algorithm must be completely evaluated. Thus, our technique

calculates a CRC (cyclic redundancy code) checksum on the bytes of the RTLs and keeps a hash table of these checksums. CRCs are commonly used to check the validity of data transmitted over a network and have an advantage over conventional checksums in that the order of the bytes of data does affect the result [67]. This property of CRC checksums is important since two function instances having the same instructions in different relative orders should be detected as distinct. If the checksum for the current function instance is the same as that generated for some previous function instance, then the two function instances are tagged as identical, and the performance results of the previous instance are used. We have verified that it is rare (we never found an instance) that the same checksum is generated for different function instances, and we never observed that the best fitness value found was affected in our experiments.

Detecting Equivalent Code

We noticed that sometimes the codes generated by different optimization sequences, although not identical, are guaranteed to perform identically in regard to speed and size. Such function instances are termed as *equivalent*. Consider two function instances that have the same sequence of instruction types, but use different registers. This situation can occur since different optimization phases compete for registers. For instance, consider the source code in Figure 4.12(a). Figures 4.12(b) and 4.12(c) show two possible translations given two different orderings of optimization phases that consume registers.

To detect this situation, we devised a technique that identifies the live ranges of all of the registers in the function and maps each live range to a distinct pseudo register. Equivalent function instances become identical after mapping, which is illustrated for the example in Figure 4.12(d). The CRC checksum for the mapped function instance is computed and checked in a separate hash table of CRC checksums to see if the mapped function had been previously generated.

On most machines there is a uniform access time for each register in the register file. Likewise, most statically scheduled processors do not generate stalls due to anti (write after read) and output (write after write) dependences. However, these dependences could inhibit future optimizations. Thus, comparing register mapped functions to avoid executions in the search should only be performed after all remaining optimizations (e.g. filling delay slots) have been applied. Given that these assumptions regarding a uniform register access time

```
sum = 0;
for (i = 0; i < 1000; i++)
   sum += a[i];
```
(a) Source Code

```
   r[10]=0;
   r[12]=HI[a];
   r[12]=r[12]+LO[a];
   r[1]=r[12];
   r[9]=4000+r[12];
L3
   r[8]=M[r[1]];
   r[10]=r[10]+r[8];
   r[1]=r[1]+4;
   IC=r[1]?r[9];
   PC=IC<0,L3;
```
(b) Register Allocation before Code Motion

```
   r[11]=0;
   r[10]=HI[a];
   r[10]=r[10]+LO[a];
   r[1]=r[10];
   r[9]=4000+r[10];
L3
   r[8]=M[r[1]];
   r[11]=r[11]+r[8];
   r[1]=r[1]+4;
   IC=r[1]?r[9];
   PC=IC<0,L3;
```
(c) Code Motion before Register Allocation

```
   r[32]=0;
   r[33]=HI[a];
   r[33]=r[33]+LO[a];
   r[34]=r[33];
   r[35]=4000+r[33];
L3
   r[36]=M[r[34]];
   r[32]=r[32]+r[36];
   r[34]=r[34]+4;
   IC=r[34]?r[35];
   PC=IC<0,L3;
```
(d) After Mapping Registers

Figure 4.12: Different Functions with Equivalent Code

and no stalls due to anti or output dependences are true, if the current mapped function is equivalent to a previous mapped instance of the function, then we can assume the two are equivalent and will produce the same result after execution.

4.2.2 Experimental Results

We applied the techniques in Section 4.2.1 to all functions in each of our benchmarks. Again we used a population size of 20 and 100 generations when attempting to find an effective optimization sequence using the genetic algorithm once for each function. Thus, 2000 optimization phase sequences are generated for each function.

Figure 4.13 shows the average number of sequences whose executions were avoided for each benchmark using the four different methods described in Section 4.2.1. Each function is weighted equally since the same number of sequences were applied for each function. The average bar is for the average of the percentages for the six benchmarks. These results do not include the functions in the benchmarks that were not executed when using the sample input data since these functions were evaluated on code size only and did not require execution of the application. As mentioned previously, each method in Section 4.2.1 is able to find a superset of the sequences handled by methods applied before it. On average 38.2% of the sequences were detected as redundantly attempted using the first technique in Section 4.2.1. 36.6% were caught as redundant active sequences using the second technique

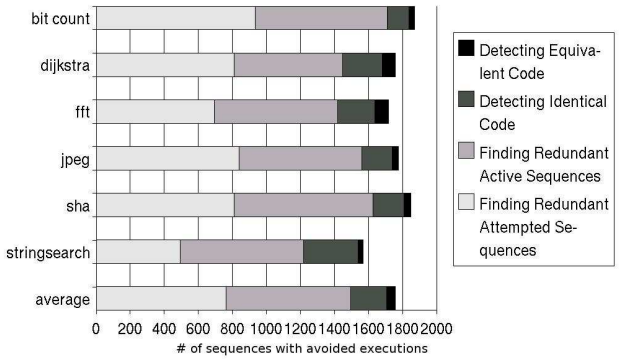


Figure 4.13: Number of Avoided Executions

in Section 4.2.1. 10.5% were discovered to produce identical code as generated by a previous sequence using the third technique, and 2.5% were found to produce unique, but equivalent code using the last technique in Section 4.2.1. Thus, over 87.7% of the executions were avoided. We discovered that sorting the phases in a sequence, so that consecutively applied independent phases are in the same order, increased the number of avoided executions by 1.15%. We found that sorting was more successful when hashing the active sequences than the attempted sequences since there was a greater chance of having a redundant sequence due to the sequence lengths being shorter after removing the dormant phases.

Figure 4.14 shows the relative search time required when applying the methods described in Section 4.2.1 to not applying these methods. These methods reduced the search time by 62%. The average time required to evaluate each of the six benchmarks improved from 6.31 hours to 2.86 hours. The reduction appears to be affected not only by the percentage of the avoided executions, but also by the size of the functions. The larger functions tended to have fewer avoided executions and also had longer compilations. While the average search time was significantly reduced for these experiments using direct execution on a SPARC processor, the savings would only increase when using simulation since the executions of the application would comprise a larger portion of the search time.

By observing the search status, as shown in Figure 4.4, we found that search progressed more quickly as the number of generations performed increased. Figure 4.15 shows the average number of redundant sequences, where execution was not required, for each of the

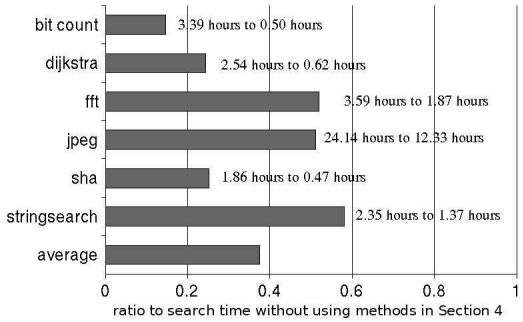


Figure 4.14: Relative Total Search Time on the SPARC

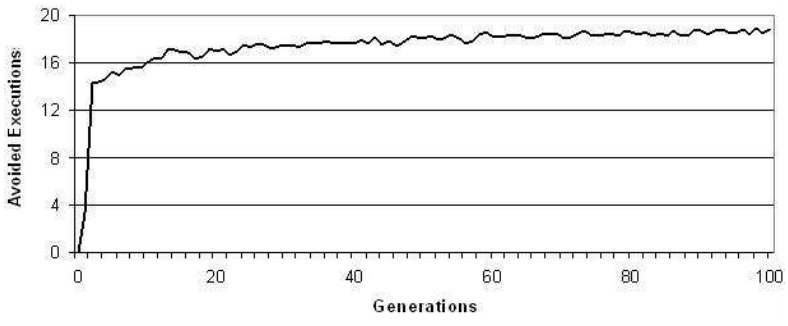


Figure 4.15: Number of Redundant Executions Avoided Per Generation

100 generations in the searches. The average number of redundant sequences generally increases as more generations are performed. This phenomenon is not surprising since there is a limited number of sequences that will produce different code. Thus, a user can double the number of generations to be performed with only a small increase in search time. Likewise, we could check for improvement for the last n generations and used this as a termination condition for the genetic algorithm.

We also found that searches performed with shorter sequences had a higher percentage of redundant executions that could be avoided. Note that the sequence length is established by the batch compiler active sequence. Smaller functions tended to have shorter sequence lengths due to fewer opportunities for optimization phases to be active. Figure 4.16 shows three plots with sequence lengths ranging from 3-10, 11-20, and 21-50. The shorter sequence

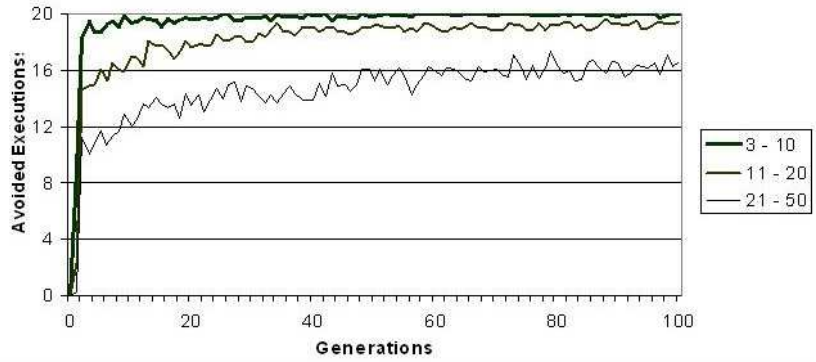


Figure 4.16: Number of Redundant Executions Avoided Per Generation for Different Sequence Lengths

lengths quickly become almost entirely redundant in a few generations. A sequence that has a shorter length is more likely to be redundant due to fewer active phases affecting the generated code. In addition, the likelihood of mutation is less when there are fewer phases in a sequence to mutate. In contrast, the longer sequences are on average much less redundant since longer sequence lengths yield more possible active sequences and more possible ways in which the final code can be generated. All three plots show that the search finds an increasing number of redundant sequences as the number of generations increases.

Figures 4.17, 4.18, and 4.19 display information regarding the number of times an optimization phase was active. Figure 4.17 shows the average number of times that the different optimization phases were active for each sequence. One should realize that an optimization phase may not be active in a sequence since the genetic algorithm may simply not select that particular phase throughout the sequence. Also, this information does not depict the number of transformations that were applied in each active phase. However, the figure does illustrate that some optimization phases, such as instruction selection and common subexpression elimination, are much more likely to be active than other phases. In addition, some phases can be accomplished by a combination of other phases. For instance, common subexpression elimination and dead assignment elimination can often have the same effect as instruction selection. Finally, the success of phases is also affected by the code generation strategy. For instance, the front end that we used always generated intermediate

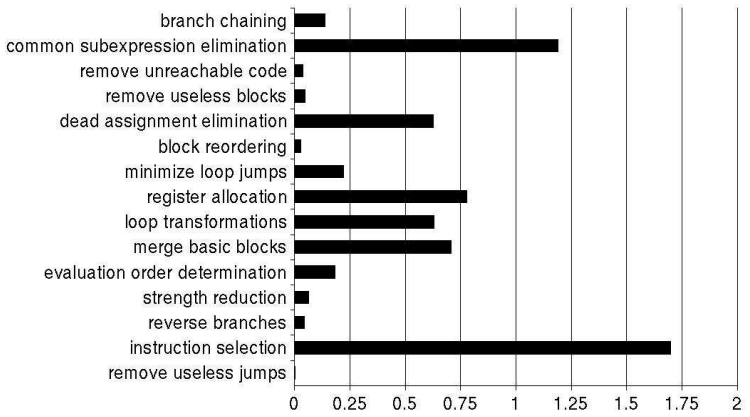


Figure 4.17: Average Times Each Phase Was Active

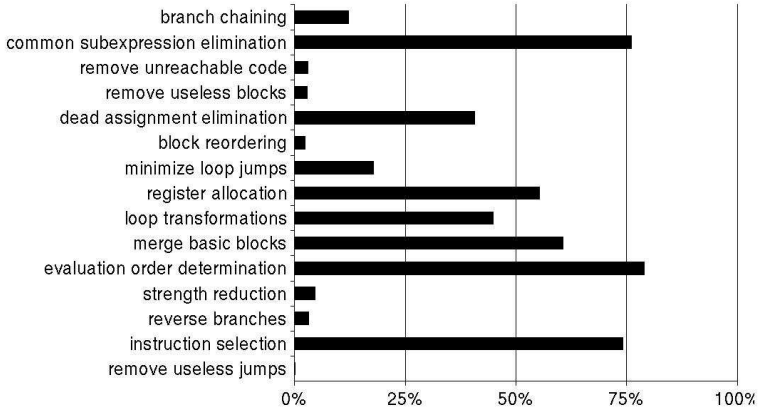


Figure 4.18: Percentage That Each Phase Was Active When Attempted

code where a label preceded the epilogue code at the end of a function in case there were return statements in the source code from other locations in the function. For functions with no conditional control flow, this return block was always merged with the entry block. Thus, the *merge basic blocks* optimization phase was successful more frequently than if another code generation strategy was used.

Figure 4.18 shows how often an optimization phase was active given that it was actually attempted. It is interesting to note that while instruction selection was the phase that was

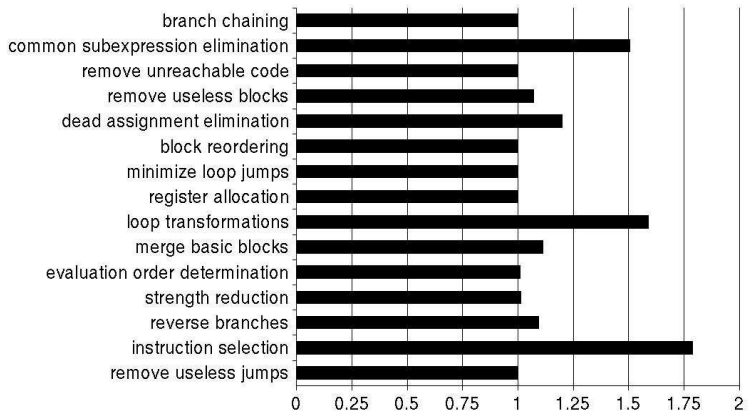


Figure 4.19: Number of Times Each Phase Was Active Given It Was Active at Least Once

active the most often, common subexpression elimination was active a greater percentage of the time when it was selected. Instruction selection has a direct impact on both code size and speed. Sometimes common subexpression elimination does not reduce code size and may not be deemed as beneficial as instruction selection by the genetic algorithm. Likewise, evaluation order determination could often be applied successfully when attempted, but had little impact on performance. The phases that did not help performance are likely to be in sequences that are in the lower half of the population. These sequences could be replaced by the crossover operation and had a higher mutation rate applied to them. Thus, phases having little impact on performance were applied less often. In addition, evaluation order determination could only be applied before assigning pseudo registers to hardware registers, which was implicitly performed before the first code-improving phase in the sequence that requires it.

Figure 4.19 shows the average number of times an optimization phase was active in a sequence given that it was active at least once. There are several optimization phases, such as branch chaining, that were active at most a single time. This shows that perhaps these phases are typically not enabled by other phases.

4.3 Producing Similar Results in Fewer Generations

Another approach that we used to reduce the search time for finding effective optimization sequences is to produce the same results in fewer generations of the genetic algorithm. We demonstrated the feasibility of this approach by developing techniques that can allow users to either specify fewer generations to be performed in their searches, or stop the search sooner once the desired results have been achieved.

4.3.1 Methods for Producing Similar Results in Fewer Generations

The following subsections describe the different techniques that we use to obtain effective sequences of optimization phases in fewer generations. All of these techniques identify phases that are likely to be active or dormant at a given point in the compilation process.

Using the Batch Sequence

The traditional or *batch* version of our compiler always attempts the same order of optimization phases for each function. We obtain the sequence of active phases (those phases that were able to apply one or more transformations) from the batch compilation of the function. We have used the length of the active batch sequence to establish the length of the sequences attempted by the genetic algorithm in previous experiments [12].

For this technique we used the active batch sequence for the function as one of the sequences in the initial population. This was based on the premise that if we initialize a sequence in the population with optimization phases that are likely to be active, then this may allow the genetic algorithm to converge faster on the best sequence it can find. This approach is similar to including in the initial population the compiler writer's manually specified priority function when attempting to tune a compiler heuristic [66].

Prohibiting Specific Phases

While many different optimization phases can be specified as candidate phases for the genetic algorithm, sometimes specific phases can never be active for a given function. If the genetic algorithm only attempts phases that have an opportunity to be active, then the algorithm may converge on the best sequence it can find in fewer attempts. There are several situations

when specific optimizations should not be attempted. Loop optimization phases cannot be active for a function that does not contain loops. Register allocation in VPO cannot be active for a function that does not contain any local variables or parameters. Branch optimizations and unreachable code elimination cannot be active for a function that contains a single basic block. Detecting that a specific set of optimization phases can never be active for a given function requires simple analysis that only needs to be performed once at the beginning of the genetic algorithm.

Prohibiting Prior Dormant Phases

When compiling a function, we realized that certain optimization phases will be dormant given that a specific prefix of active phases has been performed. Given that the same prefix of phases is attempted again, there is no benefit from attempting the same dormant phase in the same situation since it will remain dormant. To avoid repeating these dormant phases, we represent the active phases as nodes in a DAG, where each child corresponds to the next phase in an active sequence. For each node we calculate the CRC checksum for the bytes of the RTLs at that point after applying the associated optimization phase. A node in the DAG has more than one parent when different prefixes produce identical RTLs. We also stored at each node the set of phases that were found to be dormant for that prefix of active phases. Figure 4.20 shows an example DAG where the bold portions represent active prefixes and the non-bold boxes represent dormant phases given that prefix. For instance, *a* and *f* are dormant phases for the prefix **bac**. To prohibit applying a prior dormant phase, VISTA forces a phase to change during mutation until we find a phase that has either been active with the specified prefix or has not yet been attempted.

Prohibiting Unenabled Phases

Certain optimization phases when performed cannot become active again until enabled. For instance, register allocation replaces references to variables in live ranges with registers. A live range is assigned to a register when a register is available at that point in the coloring process. After the compiler applies register allocation, this optimization phase will not have an opportunity to be active again until the register pressure has changed. Unreachable code elimination and a variety of branch optimizations will not affect the register pressure and thus will not enable register allocation. Figure 4.21 illustrates that a specific phase, the

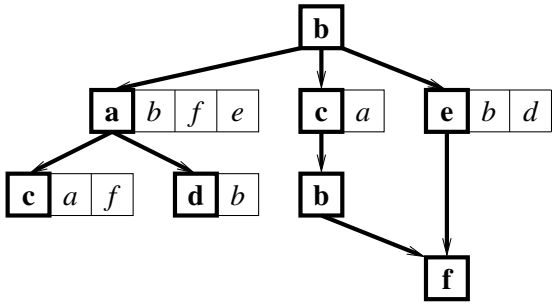


Figure 4.20: A DAG Representing Active Prefixes

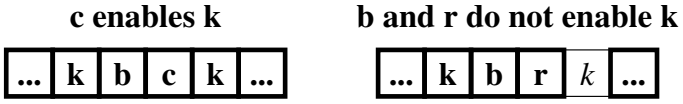


Figure 4.21: Enabling Previously Applied Phases

non-bold box of the sequence on the right, will at times be unenabled and cannot be active. Again the premise is that if the genetic algorithm concentrates on the phases that have an opportunity to be active, then it will be able to apply more active phases in a sequence and converge to the best sequence it can find in fewer attempts. Note that determining which optimization phases can enable another phase requires careful consideration by the compiler writer.

We implemented this technique by forcing a phase to mutate if the same phase has already been performed and there are no intervening phases that can enable it. We realized that a specific phase can become unenabled after an attempted phase is found to be active or dormant. We first follow the DAG of active prefixes, which was described in the previous subsection, to determine which phases are currently enabled. For example, consider again Figure 4.20. Assume that **b** can be enabled by **a**, but cannot be enabled by **c**. Given the prefix **bac**, we know that **b** cannot be active at this point since **b** was dormant after the prefix **ba** and **c** cannot reenable it. After reaching a leaf of the DAG we track which phases cannot be enabled by just examining the subsequently attempted phases.

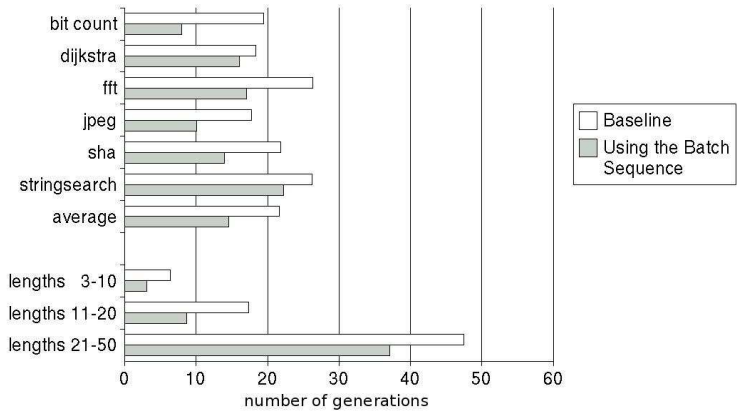


Figure 4.22: Number of Generations before Finding the Best Fitness Value When Using the Batch Sequence

4.3.2 Experimental Results

In this section we determined the average number of generations that were evaluated for each of the functions before finding the best fitness value in the search. The *baseline* result is without using any of the techniques described in Section 4.3.1. The other results indicate the generation when the first sequence was found whose performance equaled the best sequence found in the baseline search. We did not include the results for the functions when the best fitness value found was not as good as the best fitness value in the baseline, which occurred on about 3% of the functions. Not including these results caused the baseline to vary since the functions with different fitness values were not always the same when applying each of the techniques. About 9.4% of the functions had improved fitness values and about 2.8% of the functions had worse fitness values when *all* of the techniques were applied. On average the best fitness values improved by 0.04% (by 0.30% for only the differing functions). The maximum number of generations before finding the best fitness value for any function was 98 out of a possible 100 when not applying any of the four techniques. The maximum was 89 when all four techniques were used. The techniques occasionally caused the best fitness value to be found later, which we believe is due to the inherent randomness of using a genetic algorithm. However, all of the techniques were beneficial on average.

Figure 4.22 shows the effect of using the batch sequence in the initial population, which in

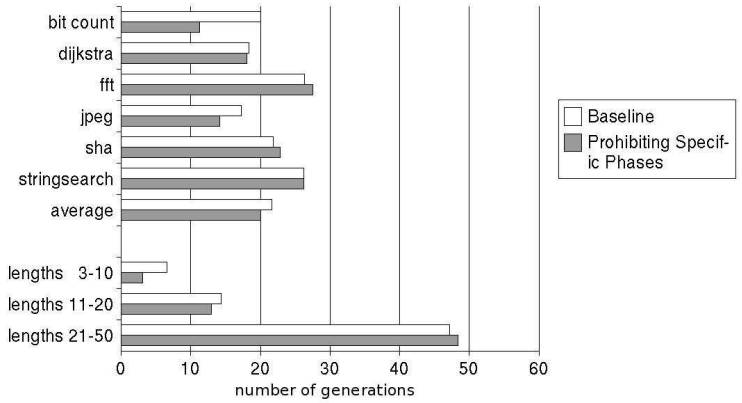


Figure 4.23: Number of Generations before Finding the Best Fitness Value When Prohibiting Specific Phases

general was quite beneficial. The last three bars show the average effect when separating the benchmarks according to the sequence length used in the search. Note that sequence length for each function is established by multiplying the active sequence of the batch compiler by 1.25. We found that this technique worked well for the smaller functions in the applications since it was often the case that the batch compiler produced code that was as good as the code generated by the best sequence found in the search. However, the smaller functions tended to converge on the best sequence in the search in fewer generations anyway since the sequence lengths were typically shorter. In fact, it is likely that performing a search for an effective optimization sequence is in general less beneficial for smaller functions since there is less interplay between phases. Using the batch sequence for the larger functions often resulted in finding the best sequence in fewer generations even though the batch compiler typically did not produce code that was as good as produced by the best sequence found in the baseline results. Thus, simply initializing the population with one sequence containing phases that are likely to be active is quite beneficial.

The effect of prohibiting specific phases throughout the search was less beneficial, as shown in Figure 4.23. Specific phases can only be safely prohibited when the function is relatively simple and a specific condition (such as no loops, no variables, or no unconditional jumps) can be detected. Several applications, such as *stringsearch*, had no or very few functions that met these criteria. The simpler functions also tended to converge faster to

the best sequence found in the search since the sequence length established by the length of the batch compilation was typically shorter. Likewise, the simpler functions also have little impact on the size of the entire application and have little impact on speed when they are not frequently executed.

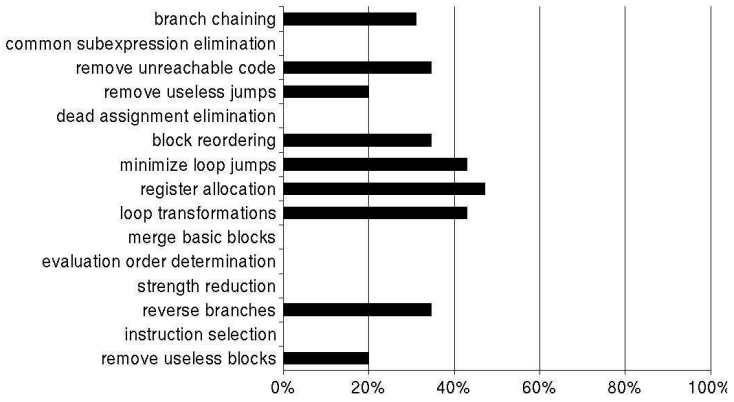


Figure 4.24: Percentage of Functions Where Each Phase Could be Prohibited

Figure 4.24 shows how often each type of phase could be prohibited. Several transfer of control optimization phases could be prohibited when the function had no such instructions. Minimize loop jumps and loop transformations could be prohibited when there were no loops in a function. Register allocation could be prohibited for only very simple functions that referenced no local variables or arguments. Several optimization phases were never prohibited since these phases could either be commonly performed or the analysis to determine they could not be applied was difficult to accomplish. In contrast, prohibiting prior dormant and unenabled phases, which are depicted in Figures 4.25 and 4.26, had a more significant impact since these techniques could be applied to all functions. Without using these two techniques, it was often the case that many phases were reattempted when there was no opportunity for them to be active.

Applying *all* the techniques produced the best overall results, as shown in Figure 4.27. In fact, only about 41% of the generations on average (from 21.38 generations to 8.85 generations) were required to find the best sequence in the search as compared to the baseline. As expected, applying all of the techniques did not result in the sum of the benefits of the individual techniques since some of the phases that were prohibited would be caught by

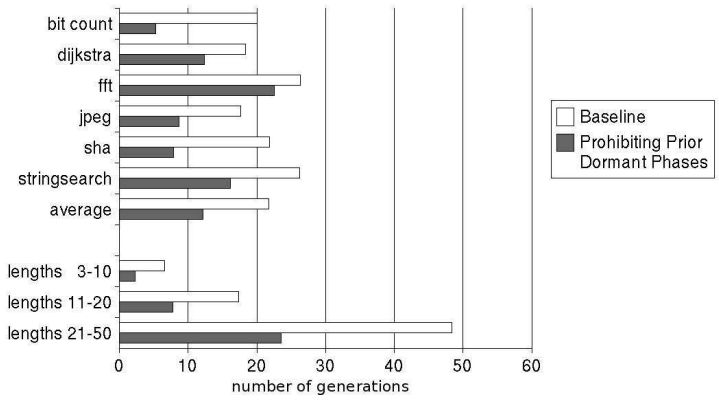


Figure 4.25: Number of Generations before Finding the Best Fitness Value When Prohibiting Prior Dormant Phases

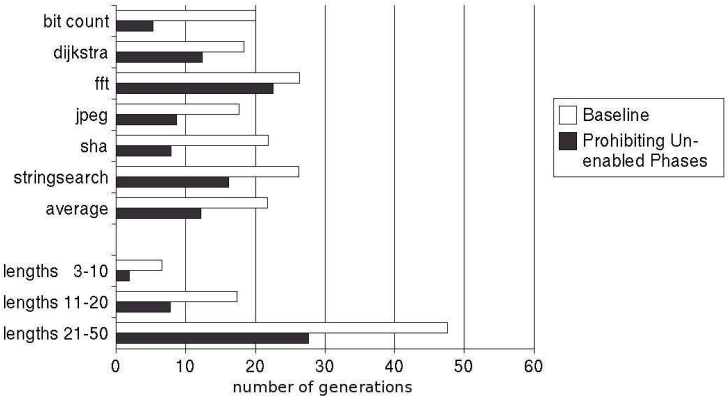


Figure 4.26: Number of Generations before Finding the Best Fitness Value When Prohibiting Unenabled Phases

multiple techniques.

Only applying phases that are likely to be active affects the number of avoided executions, which is depicted in Figure 4.28 The top bar shows the results given in Figure 4.13 from applying only Section 4.2.1 techniques. The bottom bar for each benchmark shows the number of executions that are avoided when all of the techniques described in Section 4.3.1 are applied. No active sequences were considered redundant after applying the technique described under the heading *Prohibiting Prior Dormant Phases* in Section 4.3.1 since we

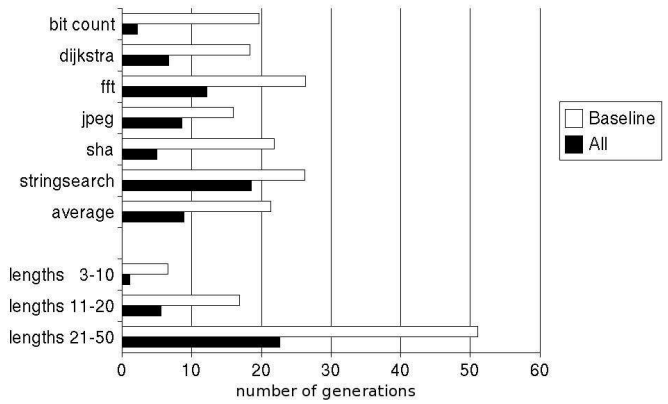


Figure 4.27: Number of Generations before Finding the Best Fitness Value When Applying All Techniques

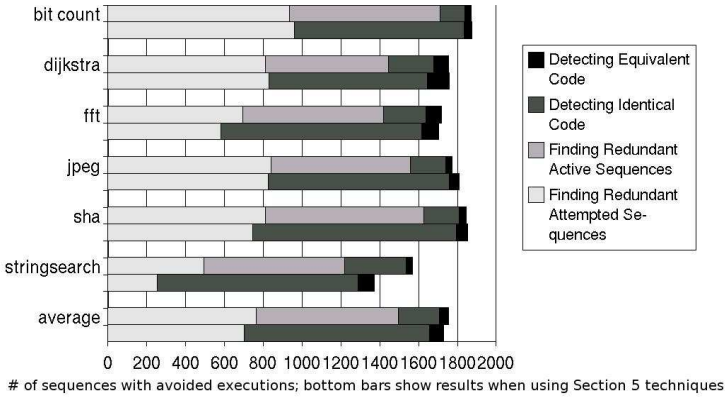


Figure 4.28: Number of Avoided Executions When Using Section 4.3.1 Techniques

checked the checksums stored in the DAG of active prefixes to determine if the active sequences produced identical code. Thus, detecting sequences as identical also detects redundant active sequences. One can see that the number of redundantly attempted sequences decreased on average. We found that many of the smaller functions had more hash table hits for attempted sequences after applying the techniques in Section 4.3.1 and the larger functions typically had fewer hits. We believe this phenomenon is due to applying the techniques to prohibit prior dormant and unenabled phases. For the smaller functions

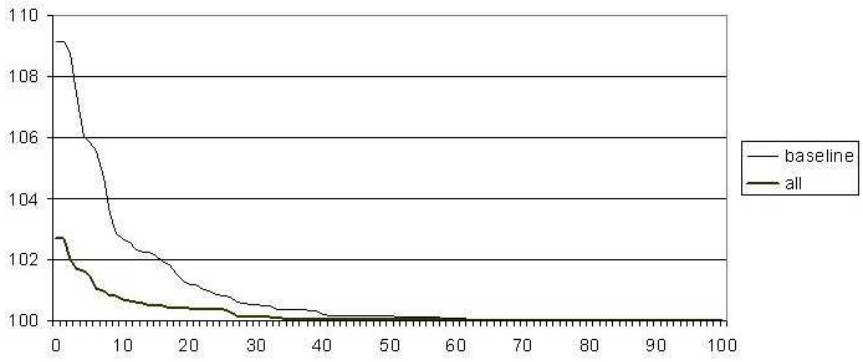


Figure 4.29: Average Benefit Relative to the Best Fitness Value Per Generation

with shorter sequence lengths, the possible phases to attempt were often exhausted and an active phase that was used before was often attempted. Likewise, the larger functions with longer sequence lengths and significantly larger search spaces tended to not reattempt previously dormant phases, but did not exhaust the possible phases and had fewer hits in the hash table. The average number of avoided executions decreases by about 1.4%, which means a greater number of functions with unique code were generated. However, the decrease in avoided executions is much less than the average decrease in generations required to reach the best sequence found in the search, as shown in Figure 4.27.

Figure 4.29 shows the impact that applying all of the techniques in Section 4.3.1 had on the average performance of the code for each generation relative to the best fitness value found in the search. A significant improvement is obtained by performing the batch sequence in the initial generation. After a few generations, prohibiting prior dormant phases and prohibiting unenabled phases result in a greater benefit than using the batch sequence. Performing all of the techniques resulted in the best result. This graph shows that the number of generations could be reduced with a negligible loss in performance of the generated code.

Figure 4.30 shows the relative time for finding the best fitness value when all of the techniques in Section 4.3.1 were applied. The actual times are shown in minutes since finding the best sequence is accomplished in a fraction of the total generations performed in the search. Note the baseline for finding the best fitness value includes all of the methods described in Section 4.2.1 to avoid unnecessary executions. The best fitness value was found

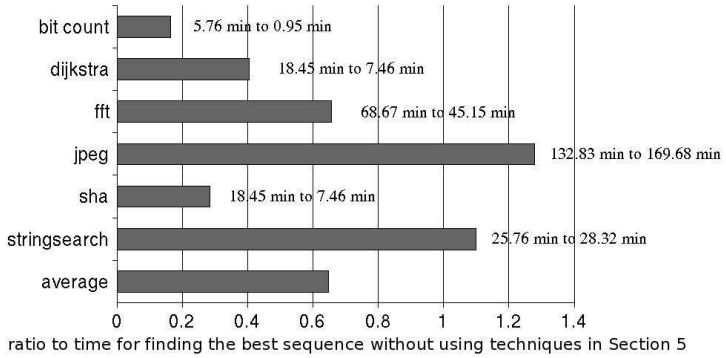


Figure 4.30: Relative Search Time before Finding the Best Fitness Value

in 65.0% of the time on average as compared to the baseline.

4.4 Implementation Challenges

During the process of this investigation, we encountered several implementation issues that made this work challenging. In this section I have attempted to document some of the challenges specific to this period of my research.

The biggest challenge for iterative compilers is producing code that always generates the correct output for different optimization phase sequences. Even implementing a conventional compiler that always generates code that produces correct output when applying one predefined sequence of optimization phases is not an easy task. In contrast, generating code that always correctly executes for thousands of different optimization phase sequences is a severe stress test for the compiler. In addition to making the implementation of our optimization phases more robust, we also had to carefully determine exactly what analysis is needed for each optimization phase, and what analysis can be invalidated by each phase. This was mainly done to improve the efficiency of our searches, since invalidating and re-performing all analysis between all phases will be very wasteful. Ensuring that all sequences in the experiments produced valid code required tracking down many errors that had not yet been discovered in the VISTA system.

Additionally, determining which phases were independent (see Table 4.1), prohibiting specific phases (see Section 4.3.1), and prohibiting unenabled phases (see Section 4.3.1)

required analysis and judgment by the compiler writer to determine when optimization phases could be enabled or disabled. We inserted sanity checks when running experiments without using these methods to ensure that our assertions concerning the enabling of optimization phases were accurate. For instance, we checked that the attempted and active sequences for every function produced the same code when applied directly or when applied after sorting the independent phases. We found several cases where our reasoning was faulty after inspecting the situations uncovered by these sanity checks and we were able to correct our enabling assertions.

We sometimes found that the optimization phases we classified as being dormant did have unexpected side effects by changing the analysis information, which could enable or disable a subsequent optimization phase. These side effects can affect the results of eliminating redundant active sequences in Section 4.2.1, and prohibiting prior dormant and unenabled phases in Section 4.3.1. We also inserted sanity checks to ensure that different dormant phases did not cause different effects on subsequent phases. We detected when these situations occurred, properly set the information about what analysis is required and invalidated by each optimization phase, and now rarely encounter these problems.

Finally, these experiments were quite time-consuming, particularly when obtaining a baseline without using our techniques to reduce the search overhead. We modified the system to log information during the search, such as each attempted sequence, the corresponding active sequence, the checksum of the function produced by the sequence, and the effect on speed and space. In order to reduce the time required to isolate problems when performing various sanity checks, we would process the log file rather than rerunning the entire search. We feel that the overhead of logging information was more than justified by the time we saved during debugging our techniques. It was, of course, possible to easily turn off logging to improve efficiency during the final experimental runs.

4.5 Concluding Remarks

There are several contributions from this work. First, we have shown that there are effective methods to reduce the search overhead for finding effective optimization phase sequences by avoiding expensive executions or simulations. Detecting when a phase was active or dormant by instrumenting the compiler was very useful since many sequences can be detected as redundant by memoizing the results of active phase sequences. We also discovered that the

same code is often generated by different sequences. We demonstrated that using efficient mechanisms, such as a CRC checksum, to check for identical or equivalent functions can also significantly reduce the number of required executions for an application. Second, we have shown that on average the number of generations required to find the best sequence can be reduced by over two thirds. One simple, but effective technique is to insert the active sequence of phases from the batch compilation as one of the sequences in the initial population. We also found that we could often use analysis and empirical data to determine when phases could not be active. These techniques result in faster convergence to more effective sequences, which can allow equally effective searches to be performed with fewer generations of the genetic algorithm.

An environment to tune the sequence of optimization phases for each function in an embedded application can be very beneficial. However, the overhead of performing searches for effective sequences using a genetic algorithm can be quite significant and this problem is exacerbated when performance measurements for an application are obtained by simulation or on a slower embedded processor. Many developers are willing to wait for tasks to run overnight to improve a product, but are unwilling to wait longer. We have shown that the search overhead can be significantly reduced, perhaps to a tolerable level, by using methods to avoid redundant executions and techniques to converge to the best sequence it can find in fewer generations.

4.6 Influence on Future Direction of Research

The results and observations we gathered during the experiments described in this chapter proved very significant in charting the course for my future research. In the final section of this chapter, I will attempt to explain some of our considerations resulting from the influence of this work.

4.6.1 Exhaustive Searches for Optimal Phase Ordering

Exhaustively evaluating the performance of all possible orderings of optimization phases had always been considered impossible over all the phases in a mature optimizing compiler. Before our present research, even we considered the task highly improbable. Optimizing compilers typically contain numerous different optimization phases (VPO has 15 different

phases). Many compilers enforce few restrictions on the ordering of optimization phases. Additionally, optimizations can potentially enable each other, implying that many phases can be active multiple times in the same sequence. The possibility of phase repetition also means that it is impossible to determine the length of the optimal phase sequence.

Our techniques for eliminating redundant optimization sequences from Section 4.2.1 demonstrated significant redundancy in the optimization phase order space. We were able to avoid execution for over 87% of the sequences we encountered. However, these results were for a heuristic genetic algorithm. Heuristic algorithms only scan a very small portion of the phase order space. They also simplify the problem by fixing the length of the optimization phase sequence for each function. Even so, the amazing redundancy was hard to ignore. Can we extend our pruning techniques to exhaustive searches as well? Seeking an answer to this question we set out to demonstrate the feasibility of exhaustive enumeration of the entire phase order space over many functions, something that was never thought possible.

4.6.2 Applying the Techniques on an Embedded Processor

After ensuring that the techniques we developed to improve the search time for effective sequences were sound, we decided to port our infrastructure to an embedded environment. This is because targeting applications in the embedded domain was very important and attractive for the techniques we developed during the current research. Embedded applications routinely need to deal with stringent constraints on performance, and hence designers are generally willing to spend the extra time and effort, as required by iterative compilation, to obtain the best possible performance.

To evaluate the performance benefits of iterative compilation on embedded systems, we obtained the results for our different performance configurations on the Intel StrongARM SA-110 processor. Figures 4.31, 4.32, and 4.33 show the percentage improvement when optimizing for speed only, size only, and 50% for each factor, respectively. Figure 4.33 shows the percentage improvement for size and speed of the generated code. The improvements on average are more significant as compared to the improvements for the SPARC, which is shown in Figure 4.7. We believe this is in part due to the ARM architecture having fewer registers. Several optimization phases allocate registers and which phase can most effectively use a register would depend on the function being compiled. In addition, the ARM has complex and somewhat unusual addressing modes that can cause unexpected tradeoffs when applying

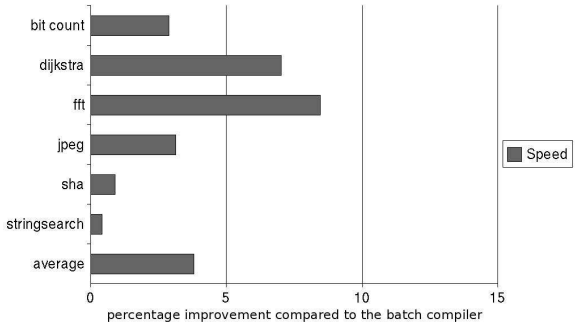


Figure 4.31: Speed Only Improvements for the ARM

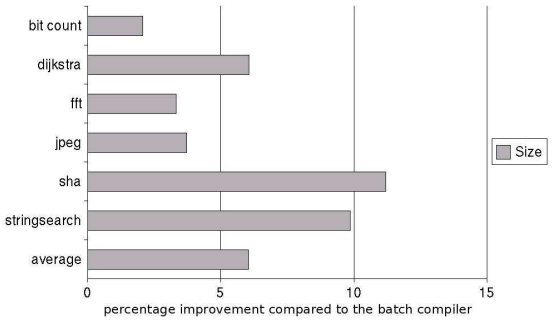


Figure 4.32: Size Only Improvements for the ARM

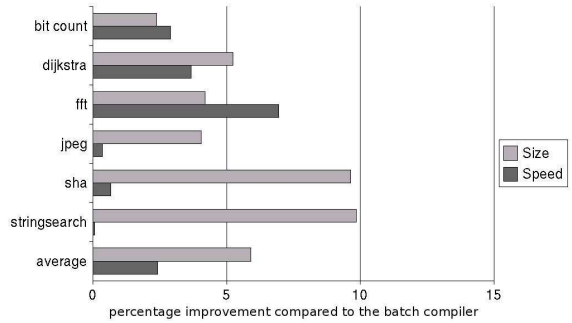


Figure 4.33: Size and Speed Improvements for the ARM

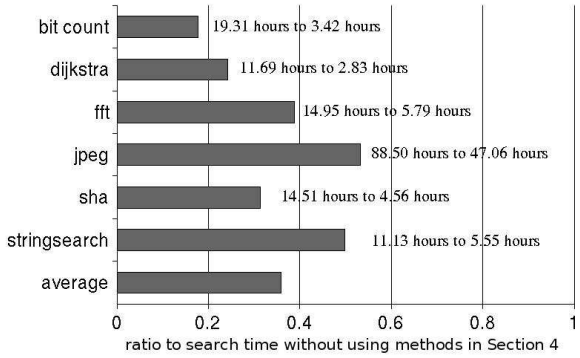


Figure 4.34: Relative Total Search Time on the ARM

optimization phases.

Figure 4.34 shows the relative time for running the genetic algorithm on the ARM when all of the techniques in Section 4.2.1 were applied. The search time using the Section 4.2.1 techniques required 35.9% of the time on average as compared to not applying these techniques. The average time required to obtain results for each of the benchmarks when optimizing for both speed and size on the ARM required 11.54 hours instead of 26.68 hours. We knew that demonstrating our results on an embedded platform will make our techniques more easily useful and more widely accepted.

4.6.3 Renouncing VISTA

The VISTA framework provided an interactive compilation environment. This meant that VISTA was specifically designed so that a user could interactively make selections using a mouse for each task. Although a great environment for debugging applications, manually improving program performance, and teaching the internals of compiler optimizations, the inherent support for interactive display was proving to be too inefficient for our needs. We had already set up a mode in VISTA where selections could be specified in a file (at the viewer end) so that the experiments could be performed in a batch mode. For such batch operations, we automatically suspended the communication of all intermediate messages between the compiler and the viewer. We eventually decided that supporting the interactive paradigm during exhaustive searches may prove to be inefficient, and unduly complicated.

Hence, for my remaining research we only concentrated our efforts on the compiler side of VISTA.

CHAPTER 5

Exhaustive Optimization Phase Order Exploration and Evaluation

During our earlier experiments using genetic algorithms we had discovered significant redundancy in the phase ordering space for all studied functions. Using various redundancy detection techniques, we found that over 87% of the sequences generated by our genetic algorithm produced identical or equivalent code. This observation allowed us to significantly speed up our genetic algorithm to find effective optimization phase sequences. In this chapter I will explain how we were able to extend our techniques to make it possible to exhaustively search the phase order space to find the best phase ordering for most of our benchmark functions. It should be noted that all references to the best (*optimal*) phase ordering (function instance) in this dissertation are with respect to the possible phase orderings in VPO, and over the benchmarks and input data set that we study during our experiments. Other compilers with a different set of optimization phases, or the same benchmarks with a different input data set can result in a different best phase ordering than the one we find.

5.1 Additional Challenges

Even with most of our techniques in place to detect massive phase order space redundancies, achieving exhaustive phase order space evaluation requires a different approach for addressing the phase ordering problem. In this section I will mention some of the issues that required a reconsideration.

Before proceeding further to attempt exhaustive phase order search space enumeration, one obvious question is: exactly how large is the *attempted* phase order space? Our compiler contains 15 different optimization phases. If each phase is attempted only once, then there are 15! = 1,307,674,368,000 possible permutations of optimization phases. Factoring in the

87% redundancy we had discovered earlier will still require 169,997,667,840 phase sequence evaluations per function. Exhaustive searches still seem impossible unless we can boost our knowledge regarding redundant phase orderings to not even attempt a considerable number of subsequent phase sequences.

Additionally, phases can enable each other. Consequently, we cannot restrict each phase to be attempted only once in each phase sequence. In fact, during our initial experiments we observed that many phases (such as *instruction selection* and *common subexpression elimination*) are active multiple times in each sequence. Allowing phase repetition for a sequence length of 15 will require us to evaluate $15^{15} = 437{,}893{,}890{,}380{,}859{,}375$ different phase sequences to find the best phase order for each function. Moreover, on allowing phase repetition we can no longer restrict the sequence length to only 15. In fact, to find the best phase ordering we cannot limit the sequence length to any fixed number. Thus, our exhaustive search algorithm will need to be independent of the sequence length.

At the same time, all the stages in any exhaustive search algorithm we develop need to be optimized to conserve both space and time. Thus, the main challenges in designing a good exhaustive phase order search algorithm will be to program the search to apply previous knowledge about redundant sequences to find additional redundancy, to make the entire process independent of the sequence length, and to optimize all stages of the algorithm for speed and memory.

5.2 Re-interpretation of the Phase Ordering Problem

Explicit listing and evaluation of all possible phase orderings cannot provide a good solution for two main reasons: (1) we do not a priori know the most appropriate sequence length to use for each function, and (2) we did not know of any way to effectively utilize previously found redundancy in the phase order space to eliminate future orderings. One crucial observation about optimization phase orderings is that each distinct ordering can only produce one distinct version of the current function (we call this a *function instance*). At the same time, since many different orderings produce the same function instance, the number of distinct function instances is clearly much smaller than the set of all possible optimization phase orderings (of different sequence lengths). Thus, rather than explicitly enumerating all possible optimization phase orderings, if we concentrate on generating all the distinct function instances (that can be produced by any phase orderings) then the problem should

become more manageable. This realization was instrumental in our success at exhaustive phase order evaluation for most of the functions we studied.

Based on this intuition, we provided a subtle re-interpretation of the phase ordering problem. We rephrased the phase ordering problem as *the problem of generating and evaluating all possible distinct function instances that can be produced by any ordering of optimization phases in our compiler for each function.* Our re-interpretation at once made the phase ordering problem, long considered intractable, much more malleable for a complete solution. The next challenge is to design a new algorithm directed at generating distinct function instances, and to find accurate and efficient methods to detect identical function instances.

5.3 Algorithm for Exhaustive Phase Order Space Evaluation

In this section I will describe the algorithm we developed to quickly and accurately evaluate all possible distinct function instances for each function. For each function, our algorithm starts with the unoptimized function instance at the root level (level 0). For level 1, we generate the function instances produced by an optimization sequence length of 1, by applying each optimization phase individually to the base unoptimized function instance. For all other higher levels, optimization phase sequences are generated by appending each optimization phase to all the sequences at the preceding level. Note that for each level n, we in effect generate all combinations of optimizations of length n. Thus, our algorithm represents the naive optimization phase order space in the form of a tree. Figure 5.1 illustrates the phase order space tree for four distinct optimization phases. Nodes in this tree represent function instances, and edges represent transitions from one function instance to another on application of an optimization phase. As can be seen from Figure 5.1, this space grows exponentially at each level and would very quickly become infeasible to traverse.

This exponentially growing search space can often be made tractable without losing any information by using many different pruning techniques. Some of these techniques have already been introduced earlier in Section 4.2.1. In the remaining part of this section, I will describe all our space pruning techniques, emphasizing their differences with our previous implementation wherever relevant.

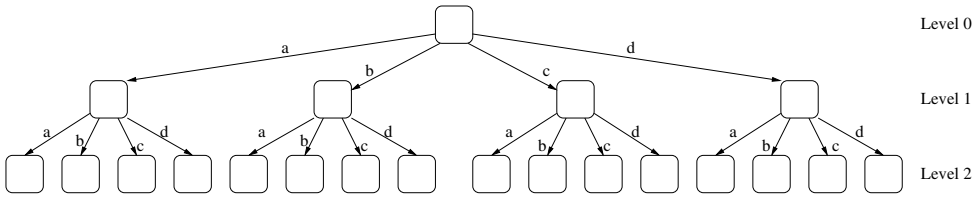


Figure 5.1: Naive Optimization Phase Order Space for Four Distinct Optimizations

5.3.1 Eliminating Consecutively Applied Phases

There is no phase in our compiler that can be successfully applied more than once consecutively. For example, consider *register allocation* being applied twice consecutively. On its first application *register allocation* will assign registers to variable live ranges as long as there are unassigned live ranges, or until we run out of free registers. On its next consecutive application, register allocation will remain dormant since the earlier terminating condition (no unassigned live ranges, or no free registers) will still be valid. Therefore, an active phase at one level is not even attempted at the next level. The phase order space after eliminating consecutively applied phases can be represented as shown in Figure 5.2.

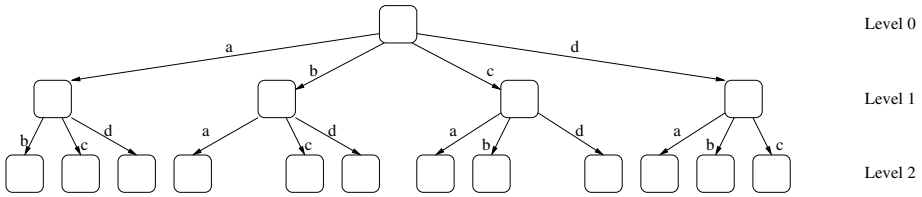


Figure 5.2: Effect of Eliminating Consecutively Applied Phases on the Search Space in Figure 5.1

5.3.2 Detecting Dormant Phases

The second pruning technique exploits the fact that not all optimization phases are successful at all levels and in all positions. As already explained, phases that are unable to find opportunities to modify the function representation are called *dormant* phases. In order to detect dormant phases we modified the compiler to provide feedback reporting if the phase was active or dormant. Since dormant phases keep the function unchanged, we do not need

to again add a node corresponding to this instance to the tree at the current level. Detecting dormant phases eliminates entire branches of the tree in Figure 5.1. The search space taking this factor into account can be envisioned as shown in Figure 5.3. The optimization phases found to be inactive are shown by dotted lines.

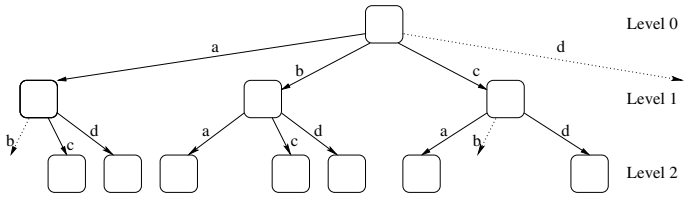


Figure 5.3: Effect of Detecting Dormant Phases on the Search Space in Figure 5.1

5.3.3 Detecting Identical Function Instances

The next pruning technique relies on the assertion that many different optimizations at various levels produce function instances that are identical to those already encountered at previous levels or those generated by previous sequences at the same level. We have already considered some reasons why different optimization sequences would produce the same code. One reason is that some optimization phases are inherently independent. As opposed to using a statically generated table of *independent phases*, as described in Section 4.2.1, for exhaustive searches we generate the code for all distinct active sequences, and explicitly check for uniqueness. We have also explained cases where sequences of different active phases can lead to the same code. If the current function instance is detected to be identical to some earlier function instance, then it can be safely eliminated from the space of distinct function instances. In other words, we do not add a new node at the current level for identical function instances.

To detect identical function instances we potentially need to compare each newly generated function instance with all previously generated distinct function instances for a match. A search may result in thousands of unique function instances, which may be too large to store in memory and very expensive to access on disk. A per character comparison would also significantly slow the search algorithm. Therefore, to make the comparisons efficient, we calculate multiple hash values for each function instance and compare the hash

values for a match. For each function instance we store three numbers: a count of the number of instructions, byte-sum of all instructions, and the CRC (cyclic-redundancy code) checksum on the bytes of the RTLs in that function. This approach has also been described in our earlier experiments using genetic algorithms in Section 4.2.1. CRCs are commonly used to check the validity of data transmitted over a network and have an advantage over conventional checksums in that the order of the bytes of data does affect the result [67]. CRCs are useful in our case since function instances can be identical except for different order of instructions. We have verified that when using all the three checks in combination it is extremely rare (we have never encountered an instance) that distinct function instances would be detected as identical.

5.3.4 Detecting Equivalent Function Instances

In earlier sections I have also explained how we detect equivalent function instances. Two different non-identical function instances are termed to be equivalent if they are guaranteed to result in identical performance. Equivalent function instances can occur since different optimization phases compete for registers. It is also possible that a difference in the order of optimizations may create and/or delete basic blocks in different orders causing them to have different labels. An example of equivalent function instances was shown in Figure 4.12. If we encounter multiple identical or equivalent function instances, then we only need to maintain one instance from each such group of function instances, and the rest can be eliminated from the search space.

In Section 4.2.1, we only check for equivalence after applying all optimization phases in their respective sequences, i.e. we are certain that no additional optimizations will be attempted on either function instance. However, during the exhaustive search algorithm we check for function equivalence during each level of the search. Therefore, we need to make our equivalence check more conservative to account for additional future optimization phases that can be later applied to the current function instance. Thus, our new check for function equivalence also guarantees that all future optimization phases will have identical effect on both instances.

To detect equivalent function instances we map each register and block label-number to a different number depending on when it is encountered in the control flow. Note that this mapping is only performed for the checksum calculation and is not used when additional

phases are applied. We start scanning the function from the top basic block. Each time a register is first encountered we map it to a distinct number starting from 1. This register would keep the same mapping throughout the function. For instance, if register `r[10]` is mapped to `r[1]`, then each time `r[10]` is encountered it would be changed to `r[1]`. If `r[1]` is later found in some RTL, then it would be mapped to the remap number existing at that position during the scan. Thus, our check for equivalent function instances is different from register remapping of live ranges as described in Section 4.2.1, and is in fact much more naive. This is because although a complete live range register remapping might detect more instances as being equivalent, we recognize that a live range remapping at intermediate points in an optimization phase sequence would be unsafe as it changes the register pressure which might affect other optimizations applied later.

Different phase orderings can also assign different labels to identical basic blocks in different function instances. Thus, during the function traversal we simultaneously remap block labels as well, which also involves mapping the labels used in the actual RTLs. The algorithm for detecting equivalent function instances then proceeds similarly to the earlier approach of detecting identical function instances.

The effect of eliminating identical or equivalent function instances from the search space tree is to transform the tree structure of the search space, as seen in Figures 5.1 and 5.3, to a directed acyclic graph (DAG) structure, as shown in Figure 5.4. By comparing Figures 5.1, 5.3 and 5.4, it is apparent how these three characteristics of the optimization search space help to make an exhaustive search more feasible. Note that the optimization phase order space for functions processed by our compiler is acyclic since no phase in VPO undoes the effect of another. However, a cyclic phase order space could also be exhaustively enumerated using our approach since identical function instances are detected.

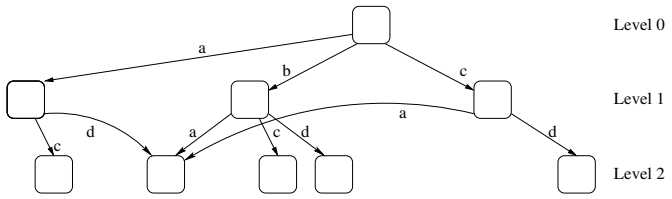


Figure 5.4: Detecting Identical Code Transforms the Tree in Figure 2 to a DAG

5.3.5 Performance Estimation of Each Distinct Function Instance

Finding the dynamic performance of a function instance requires execution or simulation of the application. Executing the program typically takes considerably longer than it takes the compiler to generate each function instance. Moreover, simulation can be orders of magnitude more expensive than native execution, and is often the only resort for evaluating the performance of applications on embedded processors. Using the techniques discussed above, we have been able to drastically reduce the phase order space so as to make it possible to enumerate all distinct function instances for each function in most cases. However, in most cases, we are still left with a substantial number of distinct function instances, which means that executing the program for each function instance in all such cases will be prohibitively expensive. Thus, in order to find the optimal function instance it is necessary to reduce the number of program executions, but still be able to accurately estimate dynamic performance for all function instances. In this section, I will describe the approach we used to achieve this.

In order to reduce the number of executions we use a technique that is based on the premise that two different function instances with identical control-flow graphs will execute the same basic blocks the same number of times. Our technique is related to the method used by Cooper et al. in their ACME system of adaptive compilation [52]. However, our adaptations have made the method simpler to implement and more accurate for our tasks. During the exhaustive enumerations we observed that for any function the compiler only generates a very small number of distinct control-flow paths, i.e. multiple distinct function instances have the same basic block control-flow structure. For each such set of function instances having the same control flow, we execute/simulate the application only once to determine the basic block execution counts for that control-flow structure.

Thus, after generating each new function instance we compare its control-flow structure with all previously encountered control flows. This check compares the number of basic blocks, the position of the blocks in the control-flow graph, the positions of the predecessors and successors of each block, and the relational operator and arguments of each conditional branch instruction. *Loop unrolling* presents a complication when dealing with loops that contain a single basic block. It is possible for loop unrolling to unroll such a loop and change the number of loop iterations. Later if some optimization coalesces the unrolled blocks, then

the control flow looks identical to that before unrolling, but due to the changed number of iterations, the block frequencies are actually different. We handle such cases by verifying the loop exit conditions and marking unrolled blocks differently than non-unrolled code. We are unaware of any other control flow changes caused by our set of optimization phases that would be incorrectly detected by our algorithm.

If the check reveals that the control flow of the new function instance has not as yet been encountered, then before producing assembly code, the compiler instruments the function with additional instructions using EASE [62]. Upon simulation, these added instructions count the number of times each basic block is executed. The functional simulator, *sim-uop* present in the SimpleScalar simulator toolset [58] is used for the simulations. The dynamic performance for each function instance is estimated by multiplying the number of static cycles calculated for each block with the corresponding block execution counts.

For each distinct function instance we then calculate the number of cycles required to execute each basic block. The dynamic performance of each function instance can then be calculated as the sum of the products of basic block cycles times the block execution frequency over all basic blocks. We call this performance estimate our *dynamic frequency measure*. Thus, the dynamic frequency measure for some function instance can be represented as:

$$dynamic\ frequency\ measure = \sum_{i=1}^{n} bfreq_i * bcyc_i \tag{5.1}$$

where, $bfreq_i$ is the basic block execution count for block i, $bcyc_i$ is the cycle count for block i, and n is the number of basic blocks in the current function instance. As long as the basic block control-flow remains unchanged for some other function instance, *bfreq* in Equation 5.1 stays the same, and we only need to recalculate *bcyc* for the new function instance.

For the purpose of the current study, the basic block cycle count is a static count that takes into account stalls due to pipeline data hazards and resource conflicts, but does not consider order dependent events, such as branch misprediction and memory hierarchy effects. Other more advanced measures of static performance, using detailed cache and resource models, can be considered at the cost of increased estimation time. As we will show later in this paper, we have found our simple estimation method to be sufficiently accurate for our needs on the in-order ARM processor.

5.3.6 Implementation Details

In this section I will describe some implementation details relevant to our exhaustive phase order search space algorithm. The phase order space can be generated/traversed in either a breadth-first or a depth-first order. Both of these approaches can be accomplished using standard graph traversal algorithms. Each traversal algorithm has different advantages and drawbacks for our experiments. In both cases we start with the unoptimized function instance representing the root node of the DAG. In breadth-first traversal, at each stage we attempt to generate all the children of the current node. Thus, using breadth first search, nodes in Figure 5.4 would be generated in the order shown in Figure 5.5(a). Depth-first search, at each stage, attempts to recursively generate the entire subtree, starting at the current node, before proceeding on to the next sibling. Thus, depth-first search would generate the nodes in Figure 5.4 in the order shown in Figure 5.5(b).

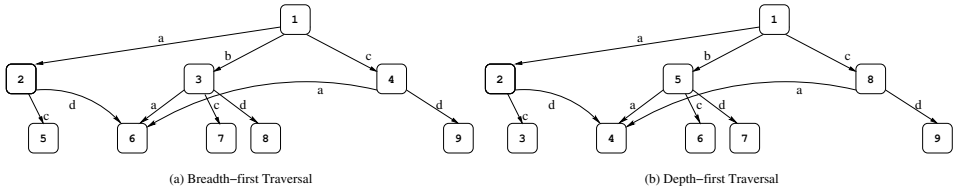


Figure 5.5: Breadth-First and Depth-First DAG Traversal algorithms

During the search process we have to compile the same function with thousands of different optimization phase sequences. Generating the function instance for every new optimization sequence involves discarding the previous compiler state (which was produced by the preceding sequence), reading the unoptimized function back from disk, and then applying all the optimizations in the current sequence. We made a simple enhancement to keep a copy of the unoptimized function instance in memory to avoid disk accesses for all optimization sequence evaluations, except the first. Another enhancement we implemented to make the searches faster uses the observation that many different optimization sequences share common prefixes. Thus, for the evaluation of each phase sequence, instead of rolling back to the unoptimized function instance every time, we determine the common prefix between the current sequence and the previous sequence, and only roll back until after the last phase in the common prefix. This saves the time that would otherwise have been required

to re-perform the analysis and optimizations in the common prefix.

To avoid re-applying optimization prefixes, we need to maintain multiple partially optimized versions of the current function at various points in the search space DAG. To re-start optimization with an old partially optimized function instance, it is essential that the internal state in the compiler be appropriately re-initialized. This is a non-trivial operation. In our earlier work, we had built the capability in VPO to throw away the current compiler state, and re-initialize the internal state by again reading in the original input file, and re-applying all previously applied transformations. For our current work, we extended this functionality by storing the current internal compiler state itself into the format of the input file. The re-initialization of the compiler to any previous compiler state now only requires us to throw away the present state as before, and read back the input file containing the required previous state. We no longer need to re-apply transformations. Moreover, by keeping all required previous compiler states in memory, this entire operation does not require any disk accesses.

Table 5.1 shows the successful sequences during the generation of the DAG in Figure 5.4 during both breadth-first and depth-first traversals. Since phases in the common prefix do not need to be reapplied, the highlighted phases in Table 5.1 are the only ones which are actually attempted. Depth-first search keeps a stack of previous phase orderings, and is typically able to exploit greater redundancy among successive optimization sequences than breadth-first search. Therefore, to reduce the search time during our experiments we used the depth-first approach to enumerate the optimization phase order search space. Also note, that the number of partially optimized function instances we need to maintain in memory at any given time is equal to the current depth level of the DAG. Since the DAGs are typically not very deep, this extra storage does not cause any significant space overhead.

There is, however, a drawback to the depth-first traversal approach. Although a large majority of the functions that we encountered can be exhaustively evaluated in a reasonable amount of time, there are a few functions, with extremely huge search spaces, which are intractable even after using all our pruning methods. It would be beneficial if we could isolate such cases before starting the search or early on in the search process so that we do not spend time and resources unnecessarily. This is easier to accomplish during the breadth-first traversal as we can see the growth in the search space at each level (refer Figure 5.4). If the growth in the first few levels is highly exponential, and difficult to tame, then we can

Table 5.1: Applied Phases during Space Traversal

	Breadth-First Traversal			Depth-First Traversal		
1.	**a**		1.	**a**	
2.	**b**		2.	a	**c**
3.	**c**		3.	a	**d**
4.	**a**	**c**	4.	**b**	
5.	a	**d**	5.	b	**a**
6.	**b**	**a**	6.	b	**c**
7.	b	**c**	7.	b	**d**
8.	b	**d**	8.	**c**	
9.	**c**	**a**	9.	c	**a**
10.	c	**d**	10.	c	**d**

stop the search on that function and designate that function as too large to exhaustively evaluate. In an earlier study, we used breadth-first search and stopped the search whenever the number of sequences to evaluate at any level grew to more than a million [8]. It is hard to find such a cut-off point during a depth-first traversal. For our final study, we stopped the exhaustive search on any function if the time required exceeded an approximate limit of 2 weeks. Please note that exhaustive phase order evaluation for most of the functions only requires a few minutes or a few hours, with only the largest enumerated functions requiring a few days.

The steps followed during the exhaustive phase order evaluation for each function are illustrated in Figure 5.6. Our depth-first search algorithm generates the next phase to apply in order to produce a new function instance. The checks to detect dormant phases, and identical/equivalent function instances are performed to remove redundant instances. If the current instance passes the redundancy checks, then we know that it is a distinct function instance, and add a corresponding node to the DAG. We then compare the basic block control-flow of the current instance with all previously encountered control-flows. If we find a match then we do not need to simulate the application, since we already know the basic block execution counts. Otherwise, we simulate an instrumented version of the application containing the current function instance, and measure the block execution counts. We then estimate the dynamic execution count for this function instance. The algorithm continues until nodes at the last level in the DAG can no longer generate any new function instances.

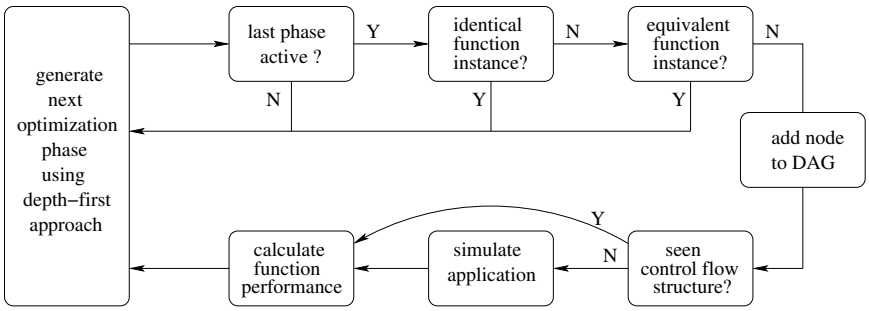


Figure 5.6: Steps followed during an exhaustive evaluation of the phase order space for each function

5.4 Experimental Results

For this study we have been able to exhaustively evaluate the phase order space for 234 out of a possible of 244 functions over 12 applications selected from the MiBench benchmark suite [59]. We selected the applications such that there are two applications from each of the six categories of benchmarks in MiBench. Only 88 out of the 244 total functions were executed when using the input data sets provided with the MiBench benchmarks. Out of the 88 executed functions, we were able to evaluate 79 functions exhaustively. We have distributed the results into two tables: Table 5.2 presents the results for the executed enumerated functions, and Table 5.3 illustrates the results for all of the enumerated functions in our benchmarks. The functions in both the tables are sorted in descending order by the number of instructions in the un-optimized function instance. Each table only presents the results for the top 50 functions in each category, along with the average numbers for the remaining functions.

The first three columns in Tables 5.2 and 5.3, namely the number of instructions, branches, and loops in the unoptimized function instance, present some of the static characteristics for each function. These numbers provide a measure of the complexity of each function. As expected, more complex functions tend to have larger search spaces. The next two columns, number of distinct function instances and the maximum active sequence length, reveal the sizes of the *actual* and *attempted* phase order spaces for each function. A maximum optimization sequence length of n gives us an attempted search space of 15^n, where 15 is the number of optimizations present in VPO. The numbers indicate that the size

Table 5.2: Dynamic Execution Count Results for all Studied Functions in the MiBench Benchmarks

Function	Inst	Br	Lp	Fn_inst	Len	CF	Leaf	within ? % of opt.			% from opt.	
								opt	2%	5%	Batch	Worst
main(t)	1275	110	6	2882021	29	389	15164	1.1	26.3	41.1	0.0	84.3
parse_sw...(j)	1228	144	1	180762	20	53	2057	0.4	1.9	4.1	6.7	64.8
askmode(i)	942	84	3	232453	24	108	475	1.7	2.9	4.6	8.4	56.2
skiptoword(i)	901	144	3	439994	22	103	2834	0.5	5.6	29.6	6.1	49.6
start_in...(j)	795	50	1	8521	16	45	80	20.0	60.0	60.0	1.7	28.4
treeinit(i)	666	59	0	8940	15	22	240	3.3	40.0	100.0	0.0	3.4
pfx_list...(i)	640	59	2	1269638	44	136	4660	0.3	0.3	2.1	4.3	78.6
main(f)	624	35	5	2789903	33	122	4214	0.0	0.0	0.0	7.5	46.1
sha_tran...(h)	541	25	6	548812	32	98	5262	0.0	9.4	30.2	9.6	133.4
initckch(i)	536	48	2	1075278	32	32	4988	24.1	91.1	91.1	0.0	108.4
main(p)	483	26	1	14510	15	10	178	1.7	21.9	32.0	7.7	13.1
pat_insert(p)	469	41	4	1088108	25	71	3021	1.4	46.6	47.3	0.0	126.4
main(j)	465	28	1	25495	21	12	134	0.0	0.0	0.0	5.6	6.0
main(l)	464	51	4	1896446	25	920	5364	0.0	25.0	25.0	0.9	89.3
adpcm_co...(a)	385	35	1	28013	23	24	230	1.3	2.6	12.6	1.8	48.9
dijkstra(d)	354	22	3	92973	22	18	1356	0.3	22.1	26.8	0.0	51.1
good(i)	313	29	1	87206	22	32	370	4.3	17.3	48.9	0.0	14.6
chk_aff(i)	304	30	1	179431	21	160	2434	1.6	10.8	39.8	0.1	58.7
cpTag(t)	303	40	0	522	11	9	16	0.0	100.0	100.0	1.6	1.6
makeposs...(i)	280	33	1	70368	24	119	498	2.4	30.5	33.7	0.0	130.1
xgets(i)	273	37	1	37960	19	103	284	4.2	32.4	32.4	0.0	129.7
missings...(i)	262	28	2	23477	26	30	513	8.2	8.2	14.4	4.0	86.8
missingl...(i)	252	31	3	11524	16	40	180	0.6	0.6	3.3	12.9	79.2
chk_suf(i)	243	21	1	75628	21	29	2835	0.8	4.4	11.7	0.8	62.4
compound...(i)	222	30	1	78429	20	49	448	3.6	3.6	3.6	11.1	100.0
main(b)	220	15	2	182246	23	84	508	0.8	16.9	16.9	8.3	250.0
skipover...(i)	212	30	1	105353	29	110	413	0.5	7.3	47.0	7.7	75.4
lookup(i)	195	22	2	37396	20	38	114	0.0	0.0	0.0	7.7	75.9
wronglet...(i)	194	25	2	22065	17	25	430	0.5	0.5	4.2	15.0	89.8
ichartostr(i)	186	26	3	40524	21	40	304	1.6	27.3	52.6	0.0	236.0
main(s)	175	12	3	30980	23	10	163	4.9	7.4	8.6	0.0	67.4
main(d)	175	15	3	9206	20	22	85	2.4	3.5	49.4	4.3	75.3
main(q)	174	14	2	38759	23	121	160	2.5	25.0	25.0	0.0	214.3
treelookup(i)	167	23	2	67507	17	65	1992	15.3	15.3	15.3	0.0	66.7
insertR(p)	161	15	0	2462	14	6	22	18.2	36.4	72.7	0.5	97.9
sha_final(h)	155	4	0	2472	13	3	68	20.6	20.6	41.2	0.0	21.7
select_f...(j)	149	21	0	510	10	10	16	25.0	25.0	75.0	0.0	7.1
byte_rev...(h)	146	5	1	2715	19	13	54	7.4	61.1	74.1	0.4	42.4
main(a)	140	10	1	1676	16	8	12	0.0	66.7	66.7	0.0	16.9
strtoichar(i)	140	18	1	10721	19	17	109	7.3	11.0	26.6	0.0	100.5
ntbl_bit...(b)	138	1	0	48	7	1	8	25.0	25.0	75.0	0.0	10.7
read_pbm...(j)	134	21	2	4182	15	18	60	6.7	16.7	20.0	6.7	69.3
bitcount(b)	133	1	0	44	8	1	7	14.3	14.3	28.6	0.0	64.3
strsearch(s)	128	17	2	32550	17	48	972	0.3	0.9	5.2	1.5	135.2
enqueue(d)	124	10	1	488	13	4	12	16.7	75.0	100.0	0.2	4.5
remaining(34)	60.1	5.0	0.5	1561.6	10.3	10.9	27.0	36.0	45.7	50.2	7.0	52.7
average(79)	234.3	21.7	1.2	174574.8	16.1	47.4	813.4	18.7	32.6	41.8	4.8	65.4

(Function - function name followed by benchmark indicator [(a)-adpcm, (b)-bitcount, (d)-dijkstra, (f)-fft, (h)-sha, (i)-ispell, (j)-jpeg, (l)-blowfish, (q)-qsort, (p)-patricia, (t)-tiff, (s)-stringsearch]), (Inst - number of instructions in unoptimized function), (Br - number of conditional and unconditional transfers of control), (Lp - number of loops), (Fn_inst - number of distinct control-flow instances), (Len - largest active optimization phase sequence length), (CF - number of distinct control flows), (Leaf - Number of leaf function instances), (within ? % of optimal - what percentage of leaf function instances are within "?"% from optimal), (% from opt - % performance difference between *Batch* and *Worst* leaf from Optimal).

Table 5.3: Code-Size Results for all Studied Functions in the MiBench Benchmarks

Function	Inst	Br	Lp	Fn_inst	Len	CF	Leaf	% from opt.	
								Batch	Worst
start_in...(j)	1371	69	2	120777	25	70	894	1.41	8.92
correct(i)	1294	106	5	1348154	25	663	7231	4.18	21.71
main(t)	1275	110	6	2882021	29	389	15164	16.25	29.12
parse_sw...(j)	1228	144	1	180762	20	53	2057	0.41	20.82
start_in...(j)	1009	55	1	39352	21	18	336	2.46	29.47
start_in...(j)	971	67	1	63458	21	30	388	1.66	6.31
askmode(i)	942	84	3	232453	24	108	475	7.87	37.08
skiptowo...(i)	901	144	3	439994	22	103	2834	1.45	24.71
start_in...(j)	795	50	1	8521	16	45	80	2.70	9.27
TeX_skip...(i)	704	77	2	5734	15	30	180	2.20	25.55
treeinit(i)	666	59	0	8940	15	22	240	2.39	5.26
pfx_list...(i)	640	59	2	1269638	44	136	4660	8.37	33.95
main(f)	624	35	5	2789903	33	122	4214	20.99	77.78
makedent(i)	555	47	2	1063697	33	70	5325	2.56	47.44
pat_remo...(p)	552	62	4	1151047	24	59	1669	0.61	63.80
sha_tran...(h)	541	25	6	548812	32	98	5262	63.77	102.90
initckch(i)	536	48	2	1075278	32	32	4988	2.80	61.54
treeinse...(i)	510	48	3	368810	26	75	1000	2.70	42.57
expandmo...(i)	493	41	2	23530	22	15	372	0.60	32.74
main(p)	483	26	1	14510	15	10	178	1.32	28.29
read_sca...(j)	480	52	2	44489	18	57	791	1.22	40.85
terminit(i)	476	33	1	3072	15	32	56	1.17	2.92
LZWReadB...(j)	472	33	2	39434	22	19	189	0.72	52.17
pat_inse...(p)	469	41	4	1088108	25	71	3021	0.00	55.71
main(j)	465	28	1	25495	21	12	134	0.00	3.97
main(l)	464	51	4	1896446	25	920	5364	9.66	35.17
dofile(i)	436	38	0	5700	16	12	136	0.00	7.33
shellesc...(i)	420	53	5	244264	21	161	3054	2.21	25.74
checkfile(i)	416	36	5	154348	24	234	2345	23.40	109.57
adpcm_co...(a)	385	35	1	28013	23	24	230	0.88	195.58
dijkstra(d)	354	22	3	92973	22	18	1356	6.59	81.32
givehelp(i)	347	10	1	584	11	9	31	1.10	5.49
usage(j)	344	1	0	34	9	1	3	0.00	2.53
TeX_math...(i)	344	49	2	69841	30	50	147	3.67	36.70
casecmp(i)	342	34	2	366006	31	37	1804	9.00	75.00
GetCode(j)	339	11	1	74531	22	20	117	4.65	9.30
show_char(i)	333	49	1	45581	17	152	812	29.63	36.42
good(i)	313	29	1	87206	22	32	370	0.00	8.00
bmhi_init(s)	309	22	4	11640	22	11	188	2.22	84.44
adpcm_de...(a)	306	28	1	44422	29	61	369	0.00	189.29
chk_aff(i)	304	30	1	179431	21	160	2434	8.16	20.41
cpTag(t)	303	40	0	522	11	9	16	6.98	6.98
update_f...(i)	294	22	1	6874	17	14	64	2.15	10.75
processC...(t)	289	31	0	1804	13	10	38	3.45	19.54
toutent(i)	286	29	1	7158	17	17	19	3.19	32.98
remaining(189)	108.7	9.9	0.7	15313.0	12.9	22.1	140.8	6.59	37.74
average(234)	196.2	17.3	1.1	89946.7	14.7	36.2	458.3	6.46	38.42

(Function - function name followed by benchmark indicator [(a)-adpcm, (b)-bitcount, (d)-dijkstra, (f)-fft, (h)-sha, (i)-ispell, (j)-jpeg, (l)-blowfish, (q)-qsort, (p)-patricia, (t)-tiff, (s)-stringsearch]), (Inst - number of instructions in unoptimized function), (Br - number of conditional and unconditional transfers of control), (Lp - number of loops), (Fn_inst - number of distinct control-flow instances), (Len - largest active optimization phase sequence length), (CF - number of distinct control flows), (Leaf - Number of leaf function instances), (% from opt - % code size difference of the *Batch* and *Worst* leaf from Optimal).

of the optimization phase order search space is 15^{16} on average, and can grow to 15^{44} (for the function *pfx_list_chk* in *ispell*) in the worst case for the compiler and benchmarks used in this study. Thus, we can see that although the attempted search space is extremely large, the number of distinct function instances is only a tiny fraction of this number. A more important observation is that unlike the attempted space, the number of distinct function instances does not typically increase exponentially as the sequence length increases. This is precisely the redundancy that we are able to exploit in order to make our approach of exhaustive phase order space enumeration feasible.

The number of distinct control flows is more significant for the performance evaluation of the executed functions in our benchmarks (Table 5.2). We only need to simulate the application once for each distinct control flow. The relatively small number of distinct control flows as compared to the total number of unique function instances makes it possible to obtain the dynamic performance of the entire search space with only relatively insignificant compile-time overhead. The next column gives a count of the *leaf* function instances. These are function instances for which no additional phase is able to make any further changes to the program representation. The small number of leaf function instances imply that even though the enumeration DAG may grow out to be very wide, it generally starts converging towards the end. It is also worthwhile noting that, for most functions, at least one leaf function instance is able to reach optimal. Note again that we are defining optimal in this context to be the function instance that results in the best dynamic frequency measures. The next three columns reveal that over 18% of all leaf instances, on average, reached optimal, with over 41% of them within 5% of optimal. We also observed that for 86.08% of the functions in our test suite at least one leaf function instance reached optimal. It is easy to imagine why this is the case, since all optimizations are designed to improve performance, and there is no optimization in VPO which undoes changes made by other optimizations before it.

We analyzed the few cases for which none of the leaf function instances achieved optimal performance. The most frequent reason we observed that caused such behavior is illustrated in Figure 5.7. Figure 5.7(a) shows a code snippet which yields the best performance and 5.7(b) shows the same part of the code after applying *loop-invariant code motion*. $r[0]$ and $r[1]$ are passed as arguments to both of the called functions. Thus, it can be seen from Figure 5.7(b) that *loop-invariant code motion* moves the invariant calculation, *r[4]+28*, out

of the loop, replacing it with a register to register move, as it is designed to do. But later, the compiler is not able to collapse the reference by *copy propagation* because it is passed as an argument to a function. The implementation of *loop-invariant code motion* in VPO is not robust enough to detect that the code will not be further improved. In most cases, this situation will not have a big impact, unless this loop does not typically execute many iterations. It is also possible that *loop-invariant code motion* may move an invariant calculation out of a loop that is never entered during execution. In such cases, no leaf function instance is able to achieve optimal phase ordering dynamic results.

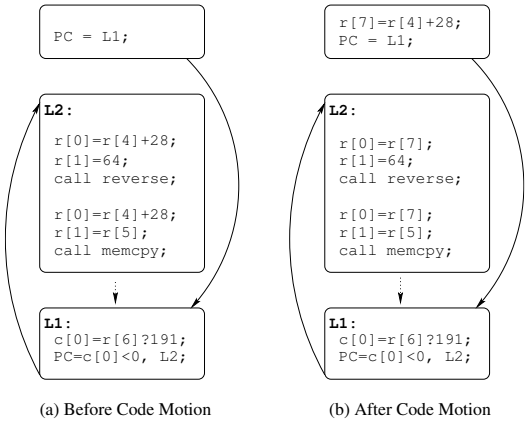


Figure 5.7: Case When No Leaf Function Instance Yields Optimal Performance

The last two columns in Table 5.2 compare the performance of the conventional (batch) compiler and the worst performing leaf function instance with the function instance(s) having the optimal ordering, with respect to our dynamic frequency measure shown in Equation 5.1. The conventional VPO compiler iteratively applies optimization phases until there are no additional changes made to the program by any phase. As a result, the fixed (batch) sequence in VPO always produces a leaf instance in our DAG. In contrast, many other compilers (including GCC) cannot do this effectively since it is difficult to re-order optimization phases in these compilers. The fixed batch optimization sequence in VPO has been tuned over several years. In spite of this aggressive baseline, the batch compiler produces code which is 4.8% worse than optimal, on average. The worst performing leaf function instance is over 65% worse than optimal on average.

The last two columns in Table 5.3 correspondingly show the code size difference between

Table 5.4: Static Features of Functions which We Could Not Exhaustively Evaluate

Function	Insts	Blk	Trans. of Cntr.		Loops		
			Cond.	Uncond.	Depth1	Depth2	Depth3
main(i)	3335	369	93	142	3	1	0
linit(i)	1842	180	51	98	5	3	0
checkline(i)	1387	203	69	96	5	2	0
save_root..(i)	1140	133	38	73	4	4	0
suf_list..(i)	823	102	33	48	1	1	0
treeoutput(i)	767	114	29	60	7	3	3
fft_float(d)	680	45	11	21	3	1	1
flagpr(i)	581	86	21	46	8	0	0
cap_ok(i)	521	95	24	54	1	5	0
prr_pre..(i)	470	65	17	33	3	0	0

(Function - function name followed by benchmark indicator [(d)-dijkstra, (i)-ispell]), (Inst - number of instructions in unoptimized function), (Blk - number of basic blocks in unoptimized function), (Trans. of Cntr. - number of conditional and unconditional transfers of control), (Loops - number of loops with nesting depths 1, 2, or 3),

the batch and the worst leaf code size, as compared to the best achievable leaf code size for each function. On average, the function instance produced by the batch compiler is 6.46% larger than the smallest leaf, with the worst leaf function instance over 38% larger than the best leaf.

Table 5.4 displays the unoptimized static features of the functions that we were unable to exhaustively enumerate according to our stopping criteria. As explained earlier, we terminate the exhaustive evaluation for any function when the time required for the algorithm exceeds two weeks. Out of the 244 possible functions, we were unable to exhaustively evaluate only 10 of those functions. It is difficult to identify one property of each function that leads to such uncontrollable growth in the phase order space. Instead, we believe that some combination of the high number of loops, greater loop nesting depths, branches, number of instructions, as well as the instruction mix are responsible for the greater phase order space size.

5.5 Correlation between Dynamic Frequency Measures and Processor Cycles

In order to make our approach of exhaustive phase order space evaluation feasible, we have avoided simulating the application to determine the performance of each distinct function instance. Instead, we have used a measure of estimated performance based partly on static

function properties. Although we have tried to account for pipeline data hazards related stalls in our estimate, it still does not consider other penalties encountered during execution, such as branch misprediction and cache miss penalties. In spite of the potential loss in accuracy, we expect our measure of performance to be sufficient for achieving our goal of finding the optimal phase ordering with only a handful of program simulations. This is especially true for embedded applications. Unlike general-purpose processors, the cycles obtained from a simulator can often be very close to executed cycles in a low-power embedded processor since these processors typically have simpler hardware and no operating system. For similar reasons, dynamic frequency measures on embedded processors will also have a much closer correlation to simulated cycles, than for general-purpose processors. It is also important to realize that it is not critical that our measure of dynamic frequencies exactly match the simulator cycles. However, what is required is a strong correlation between dynamic frequencies and simulator cycles. In this section, we perform some studies to show that this is exactly the case, at least within our test environment and experimental framework. Before listing the correlation results we first describe a modification we made to SimpleScalar cycle accurate simulator to make it faster.

5.5.1 Mixed Mode Simulator

The SimpleScalar simulator toolset [58] includes many different simulators intended for different tasks. Most useful for our purposes are the two simulators *sim-uop* and *sim-outorder*. *Sim-uop* is a functional simulator which implements the architecture, only performing the actual program execution. *Sim-outorder* is a performance simulator which implements the microarchitecture, modeling the system resources and internals in addition to executing the program. Thus, *sim-uop* is relatively fast but only provides dynamic instruction counts, whereas *sim-outorder* is able to provide processor cycles, but takes many times longer to run. In our experiments we use *sim-outorder* with the *inorder* flag for the ARM, which is generally an in-order processor for most implementations.

For our tests we only concentrate on one function at a time. Although the cache and global branch access patterns of the remaining functions in the application, as well as the library functions can affect the performance of the current function being optimized, side effects should generally be minimal. In such a scenario it would be ideal if we could run only the current function through the cycle accurate simulator, and run all remaining functions

using the faster functional simulator. This method has the potential of reducing the time required for simulations close to the level provided by the faster functional simulator, while still being able to provide accurate processor cycles for the function in question.

Sim-outorder already has a mode by which we can apply only the functional simulation for some number of initial instructions before starting the cycle accurate simulation. But once the cycle simulation was started, it was not possible to go back to the functional mode. We extended *sim-outorder* to include the ability of going back to the functional mode from the cycle mode, so that we can essentially switch back and forth between the two modes whenever desired. Before each simulator run required during our experiments, we first use the Unix *nm* utility to get the start and end instruction addresses for the current function. Later during simulation, we switch to the cycle mode only when the program counter is between this address range for the current function. Whenever the address falls out of this range we wait for the pipeline to empty and then revert back to functional simulation. This approach gives us substantial savings in time with very little loss in accuracy.

5.5.2 Complete Function Correlation

Even when using the mixed mode SimpleScalar simulator, it is very time consuming to simulate the application for all function instances in every function, instead of simulating the program only on encountering new control flows. We have simulated all instances of a single function completely to provide an illustration of the close correlation between processor cycles and our estimate of dynamic frequency counts. Figure 5.8 shows this correlation for all the function instances for the *init_search* function in the benchmark *stringsearch*. This function was chosen mainly because it is relatively small, but still has a sufficient number of distinct function instances to provide a good example.

In addition to comparing the dynamic frequency estimates and *fast* (mixed-mode) simulator cycles, Figure 5.8 also shows that the mixed-mode cycles are almost identical to the actual cycles from *sim-outorder*, verifying that our modifications did not cause much deviations from actual performance. All these performance numbers are sorted on the basis of dynamic frequency counts. Thus, we can see that our estimate of dynamic frequency counts closely follows the processor cycles most of the time. What is more important is that the correlation gets better as the function is better optimized. The excellent correlation between dynamic frequency estimates and fast/slow cycles for the optimized function instances allows

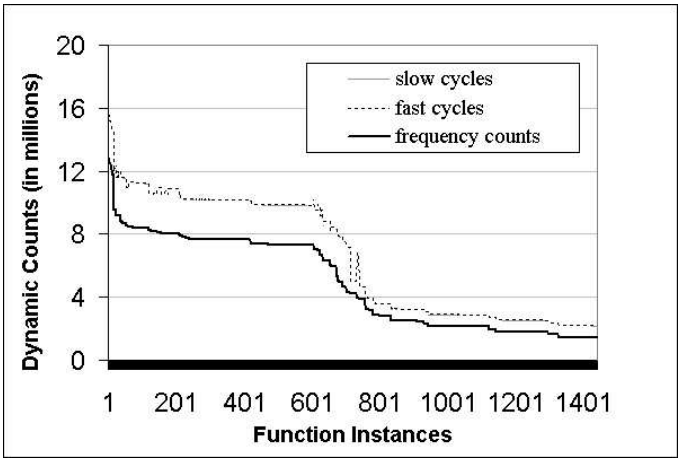


Figure 5.8: Correlation between Processor Cycles and Frequency Counts for *init_search*

us to predict the function instances with good/optimal cycle counts with a high level of confidence.

5.5.3 Correlation for Leaf Function Instances

Figure 5.9 shows the distribution of the dynamic frequency counts as compared to the optimal counts for all distinct function instances, averaged over all 79 executed functions. From this figure we can see that the performance of the leaf function instances is typically very close to the optimal performance, and that leaf instances comprise a significant portion of optimal function instances as determined by the dynamic frequency counts. From the discussion in Section 5.4 we know that for more than 86% of the functions in our benchmark suite there was at least one leaf function instance that achieved the optimal dynamic frequency counts. Moreover, it is important to note that the leaf instances constitute the only set of function instances that can be produced by the class of aggressive compilers, which includes VPO, that iteratively apply optimization phases until no additional improvements can be made.

Since the leaf function instances achieve good performance across all our functions, it is worthwhile to concentrate on leaf function instances. These experiments require hundreds of program simulations, which are very time consuming. So, we have restricted this study to only one application from each of the six categories of MiBench benchmarks. For all the executed functions from the six selected benchmarks we get simulator cycle counts for only the leaf function instances and compare these values to our dynamic frequency counts.

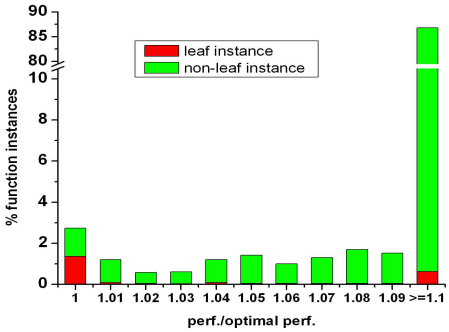


Figure 5.9: Average Distribution of Dynamic Frequency Counts

In this section we show the correlation between dynamic frequency counts and simulator cycle counts for only the leaf function instances for all executed functions over six different applications in our benchmark suite.

The correlation between dynamic frequency counts and processor cycles can be illustrated by various techniques. A common method of showing the relationships between variables (data sets) is by calculating Pearson's correlation coefficient for the two variables [68]. The Pearson's correlation coefficient can be calculated by using the formula:

$$Pcorr = \frac{\sum xy - \frac{\sum x \sum y}{n}}{\sqrt{(\sum x^2 - \frac{(\sum x)^2}{n}) * (\sum y^2 - \frac{(\sum y)^2}{n})}} \tag{5.2}$$

In Equation 5.2 x and y correspond to the two variables, which in our case are the dynamic frequency counts and simulator cycles, respectively. Pearson's coefficient measures the strength and direction of a linear relationship between two variables. Positive values of $Pcorr$ in Equation 5.2 indicate a relationship between x and y such that as values for x increase, values of y also increase. The closer the value of $Pcorr$ is to 1, the stronger is the linear correlation between the two variables. Thus, $Pcorr = +1$ indicates *perfect* positive linear correlation between x and y.

It is also worthwhile to study how close the processor cycle count for the function instance that achieves the best dynamic measure, is to the best overall cycle count over all the leaf function instances. To calculate this measure, we first find the best performing function instance(s) for dynamic frequency counts and obtain the corresponding simulator cycle count for that instance. In cases where multiple function instances provide the same best dynamic

frequency count, we obtain the cycle counts for each of these function instances and only keep the best cycle count amongst them. We then obtain the simulator cycle counts for all leaf function instances and find the best cycle count in this set. We then calculate the following ratio for each function:

$$Lcorr = \frac{best\ overall\ cycle\ count}{cycle\ count\ for\ best\ dynamic\ freq\ count} \tag{5.3}$$

The closer the value of Equation 5.3 comes to 1, the closer is our estimate of optimal by dynamic frequency counts to the optimal instance using simulator cycles.

Table 5.5 lists our correlation results for the leaf function instances over all studied functions in our benchmarks. The column, labeled *Pcorr* provides the Pearson's correlation coefficient according to Equation 5.2. An average correlation coefficient value of 0.96 implies that there is excellent correspondence between dynamic frequency counts and cycles. The next column shows the value of *Lcorr* calculated by Equation 5.3. The following column gives the number of distinct leaf function instances which have the same best dynamic frequency counts. These two numbers in combination indicate that an average simulator cycle performance of *Lcorr* can be reached by simulating only *nLf* number of the best leaf function instances as determined by our estimate of dynamic frequency measure. Thus, it can be seen that an average performance within 98% of the optimal simulator cycle performance can be reached by simulating, on average, less than 5 good function instances having the best dynamic frequency measure. The next two columns show the same measure of *Lcorr* by Equation 5.3, but instead of considering only the best leaf instances for dynamic frequency counts, they consider all leaf instances which come within 1% of the best dynamic frequency estimate. This allows us to reach within 99.6% of the optimal performance, on average, by performing only 21 program simulations per function. In effect, we can use our dynamic frequency measure to prune most of the instances that are very unlikely to achieve the fewest simulated cycles.

The conclusions of this study are limited since we only considered leaf function instances. It would not be feasible to get cycle counts for all function instances over all functions. In spite of this restriction, the results are interesting and noteworthy since they show that a combination of static and dynamic estimates of performance can predict pure dynamic performance with a high degree of accuracy. This result also leads to the observation that we should typically only need to simulate a very small percentage of the best performing

function instances as indicated by dynamic frequency counts to obtain the optimal function instance by simulator cycles.

Table 5.5: Correlation Between Dynamic Frequency Counts and Simulator Cycles for Leaf Function Instances

Function	Pcorr	Lcorr 0%		Lcorr 1%	
		Diff	nLf	Diff	nLf
AR_btbl_b...	1.00	1.00	1	1.00	1
BW_btbl_b...	1.00	1.00	2	1.00	2
bit_count	1.00	1.00	2	1.00	2
bit_shifter	1.00	1.00	2	1.00	2
bitcount	0.89	0.92	1	0.92	1
main	1.00	1.00	6	1.00	23
ntbl_bitc...	0.99	0.95	2	0.95	2
ntbl_bitcnt	1.00	1.00	2	1.00	2
dequeue	0.99	1.00	6	1.00	6
dijkstra	1.00	0.97	4	1.00	269
enqueue	1.00	1.00	2	1.00	4
main	0.98	1.00	4	1.00	4
print_path	1.00	1.00	2	1.00	2
qcount	1.00	1.00	1	1.00	1
CheckPoin...	0.95	1.00	2	1.00	5
IsPowerOf...	0.93	0.98	3	1.00	24
NumberOfB...	0.84	1.00	1	1.00	20
ReverseBits	1.00	1.00	2	1.00	2
byte_reve...	0.89	1.00	1	1.00	3
main	0.71	1.00	25	1.00	74
sha_final	0.72	0.82	26	1.00	50
sha_init	0.98	1.00	4	1.00	9
sha_print	0.95	0.88	1	1.00	6
sha_stream	1.00	1.00	1	1.00	8
sha_trans...	0.97	1.00	2	1.00	35
sha_update	0.98	1.00	14	1.00	32
finish_in...	1.00	1.00	1	1.00	1
get_raw_row	1.00	1.00	7	1.00	7
jinit_rea...	1.00	1.00	2	1.00	2
main	1.00	0.99	2	1.00	153
parse_swi...	0.95	1.00	8	1.00	16
pbm_getc	0.99	1.00	2	1.00	2
read_pbm_...	0.73	0.98	2	0.98	2
select_fi...	0.97	0.90	3	1.00	12
start_inp...	0.95	0.99	12	0.99	15
write_std...	1.00	1.00	1	1.00	1
init_search	1.00	1.00	1	1.00	14
main	1.00	1.00	8	1.00	12
strsearch	1.00	1.00	3	1.00	3
average	0.96	0.98	4.38	0.996	21

Pcorr - Pearson's correlation coefficient, Lcorr - ratio of cycles for dynamic frequency to best overall cycles (0% - optimal, 1% - within 1 percent of optimal frequency counts), Diff - ratio for *Lcorr*, nLf - number of leaves achieving the specified dynamic performance

5.6 Concluding Remarks

During our experiments with making genetic algorithms faster, we had observed that there is huge redundancy in the optimization phase order space, in the sense that many different orderings of optimization phases produce the same code. This observation led us to the realization that the space of all possible distinct *function instances* can be many orders of magnitude smaller than the space of all possible *attempted phase orderings*. In this chapter, I showed how we used this intuition to guide us in developing novel search algorithms that focus on enumerating all distinct function instances that can be produced by any phase ordering for each function. Along with new search algorithms, we also adapted all our search space pruning techniques from Section 4.2.1 to eliminate redundant function instances. We were able to exhaustively enumerate the entire phase order space for 234 out of the 244 functions that we studied.

Even after all our pruning techniques we were left with tens of thousands of unique function instances for each function. We realized that using execution or simulation to determine the performance of all instances will be prohibitively expensive. Instead, to find the optimal phase ordering performance we decided to use accurate estimations. Static phase order space enumeration had only found very few distinct basic block control flows for each function. In this chapter, I also showed how we leveraged this information to make our performance estimations more accurate. This is achieved by only simulating the application once for each set of function instances having identical control-flow structure to find their basic block execution count. The block count combined with the static block cycle count estimate gives us our *dynamic performance counts* for each function instance. We also showed that our performance estimation bears close correlation with simulator cycles. Thus, we are now able to find optimal phase orderings with our measure of dynamic performance counts, as well as find near-optimal phase orderings with respect to simulator cycles for a large majority of our functions in a reasonable amount of time.

CHAPTER 6

Analysis of the Exhaustive Optimization Phase Order Space and Applications to Exploit It

Exhaustive enumeration of the optimization phase order search space for a sizable number of functions has given us a large data set that we can analyze to gather some interesting information about optimization phase interactions, and phase order space characteristics. We can further use such information for various tasks, such as to improve conventional compilation, as well as to enhance heuristic iterative compilation techniques. This chapter will describe the results of some of our efforts to extract, study, and apply phase order space information to improve other compilation paradigms. This is, obviously, not an comprehensive list of applications that can benefit by a thorough understanding of the phase order space, but is mostly intended to serve as a primer for future exploration.

6.1 Optimization Phase Interaction Analysis

The unpredictable interaction between the various optimization phases is responsible for the phase ordering issues experienced by optimizing compilers. In this section I will present some analysis to characterize phase interactions. To assemble these statistics we first represented the search space in the form of a DAG. The nodes in the DAG represent distinct function instances and the directed edges are marked by the optimization phase that was applied from one node (function instance) to the next. This representation is illustrated in Figure 6.1. The nodes of the DAG are weighted by how many leaves it can reach. The leaf nodes have a weight of 1. The weight of each interior node is the summation of the weights of all its child nodes. Thus, the weight of each interior node gives the number of distinct sequences that are active through that point. Active phases at each node (indicated in brackets for interior nodes) in Figure 6.1 are simply the active phases that are applied on outgoing edges of that

node. We studied three different phase interactions: enabling, disabling and independence relationships between optimization phases. The following sections describe the results of this study.

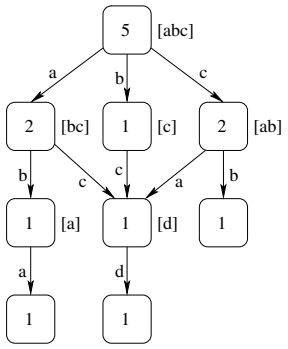


Figure 6.1: Weighted DAG Showing Enabling, Disabling, and Independence Relations

6.1.1 Enabling Interaction between Phases

A phase x is said to enable another phase y, if y was dormant (inactive) at the point just before x was applied, but then becomes active again after application of x. For example, b enables a along the path a-b-a in Figure 6.1. Note that it is possible that a phase could enable some other phase on some sequence but not on others. Thus, it could be seen that a is not enabled by b along the path c-b. Likewise, it is also possible for phases to be dormant at the start of the compilation process, and become active later (e.g., phase d along the path b-c-d). As a result we represent this information in the form of the probability of each phase enabling each other phase. We calculate the enabling probabilities by considering $dormant \rightarrow active$ and $dormant \rightarrow dormant$ transitions of phases between nodes, adjusted by the weight of the child node. The probability is the ratio of the number of $dormant \rightarrow active$ transitions to the sum of $dormant \rightarrow active$ and $dormant \rightarrow dormant$ transitions between optimization phases. We do not consider $active \rightarrow active$ and $active \rightarrow dormant$ transitions since phases already active cannot be enabled. Please note that the enabling probabilities only correspond to the phase order space information we have collected over 234 functions. We summarize the enabling information we collected for all the functions in Table 6.1, where each row represents the probability of that phase enabling other phases represented in columns. The

second column (St) in this table shows the probability over all functions evaluated, of each phase being active at the start of the compilation process.

Table 6.1: Enabling Interaction between Optimization Phases

Phase	St	b	c	d	g	h	i	j	k	l	n	o	q	r	s	u
b	0.72		0.02		0.01		0.04				0.01			0.02		0.66
c	1.00	0.01				0.68	0.01		0.02	0.07	0.05		0.15		0.34	
d			1.00				1.00									1.00
g		0.22	0.28			0.17		0.05		0.02	0.14			0.34	0.09	0.15
h	0.08		0.16						0.14	0.02	0.01				0.20	
i	0.72		0.04					0.01			0.09					
j	0.03		0.06				0.44									
k			0.98		0.28	0.01				0.02	0.01				0.96	
l	0.60		0.73			0.02			0.01		0.01			0.03	0.53	
n	0.41		0.36				0.01		0.01					0.01	0.29	
o	0.88		0.40								0.03					
q			0.99								0.02				0.99	
r	0.57		0.06								0.06					
s	1.00		0.33			0.41			0.83	0.07	0.05	0.15	0.07			
u					0.01			0.01			0.02					

Blank cells indicate an enabling probability of less than 0.005. St represents the probability of a phase being active at the start of compilation.

A few points regarding the enabling information are worth noting. The enabling interaction table is relatively sparse meaning that most phases do not enable another, which limits the maximum depth of the DAG and the total number of instances. For our benchmarks *instruction selection*(s) and *common subexpression elimination*(c) are always active initially. In contrast, *register allocation*(k) requires *instruction selection*(s) to be enabled in VPO so that the loads and stores contain the addresses of local scalars. *Instruction selection*(s) is frequently enabled by *register allocation*(k) since loads and stores are replaced by register-to-register moves, which can typically be collapsed by *instruction selection*(s). In contrast, control flow optimizations (e.g., *branch chaining*(b)) are never enabled by *register allocation*(k), which does not affect the control flow. The numbers in the table also indicate that many optimizations have a very low probability of being enabled by any other optimization phase. Such optimizations will typically be active at most once in each optimization sequence. *Remove unreachable code*(d) is rarely active for the functions in our benchmark suite, which indicates the need for a larger set of functions. Note that, unreachable code occasionally left behind by *branch chaining* is removed during *branch chaining* itself, since we found such code hindering some analysis which caused later

optimizations to miss some code improving opportunities.

6.1.2 Disabling Interaction between Phases

Another related measure is the probability of each phase disabling some other phase. This relationship can be seen in Figure 6.1 along path *b*-*c*-*d*, where *a* is active at the root node, but is disabled after *b*. The statistics regarding the disabling interaction between optimization phases is illustrated in Table 6.2. Each value in this table is the weighted ratio of *active* → *dormant* transitions to the sum of *active* → *dormant* and *active* → *active* transitions. We do not consider *dormant* → *dormant* and *dormant* → *active* transitions since a phase has to be active to be disabled.

Table 6.2: Disabling Interaction between Optimization Phases

Phase	b	c	d	g	h	i	j	k	l	n	o	q	r	s	u
b	1.00			0.28		0.09	0.18			0.20			0.11		0.01
c	0.01	1.00			0.02		0.08	0.02	0.30	0.32	1.00			0.08	
d			1.00		0.03					0.01				0.01	
g	0.13			1.00		0.06	0.01			0.12					0.22
h	0.01	0.01			1.00				0.04	0.10	1.00		0.01		
i	0.02			0.22		1.00	0.20			0.01			0.44		0.91
j				0.01		0.08	1.00						0.01		0.16
k				0.01	0.05			1.00	0.05	0.14	1.00				
l	0.02		1.00	0.11	0.04			0.07	1.00	0.32	1.00				
n	0.07	0.01		0.02	0.01		0.01			1.00	1.00				0.01
o					0.01		0.08			0.01	1.00				
q												1.00			
r	0.06					0.20	0.36						1.00		0.05
s		0.07			0.03				0.31	0.22	0.14	0.26	0.02	1.00	
u	0.41			0.02		0.34	0.15								1.00

Blank cells indicate a disabling probability of less than 0.005.

From Table 6.2 it can be seen that phases are much more likely to be disabled by themselves than by other phases. We can also see that phases such as *register allocation*(k) and *common subexpression elimination*(c) always disable *evaluation order determination*(o) since they require *register assignment*, and *evaluation order determination* can only be performed before *register assignment*. All the phase disabling values along the diagonal are equal to one, since phases in our compiler always disable themselves when attempted.

6.1.3 Optimization Phase Independence

The third interaction we measured was the probability of independence between any two optimization phases. Two phases can be considered to be independent if their order does not matter to the final code that is produced. This is illustrated in Figure 6.1 along the paths *a*-*c* and *c*-*a*. Both orders of phases *a* and *c* in these sequences produce identical function instances, which would mean that they are independent in this situation. In contrast, sequences *b*-*c* and *c*-*b* do not produce the same code. Thus, they are considered dependent in this situation. If two optimizations are detected to be completely independent, then we would never have to evaluate them in different orders. This observation can lead to the potential of even greater pruning of the search space. Table 6.3 shows the probability of each phase being independent of some other phase. This is a weighted ratio of the times two consecutively active phases produced the same code to the number of times they were consecutively active.

Table 6.3: Independence Relationship between Optimization Phases

Phase	b	c	d	g	h	i	j	k	l	n	o	q	r	s	u
b	1.00	1.00		0.09	1.00	0.84	0.99	1.00	0.95	0.59	1.00	1.00	0.73	1.00	0.55
c	1.00	1.00	0.28	0.90	0.90	0.99	1.00	0.41	0.09	0.32		0.92	0.96	0.19	0.99
d		0.28	1.00		0.80	1.00				0.84			1.00	0.93	
g	0.09	0.90		1.00	0.99	0.29	0.80	0.96	0.87	0.96		0.99	0.86	0.84	0.92
h	1.00	0.90	0.80	0.99	1.00	1.00	0.99	0.71	0.82	0.79		0.99	0.98	0.94	1.00
i	0.84	0.99	1.00	0.29	1.00	1.00	0.72	0.99	0.88	0.99	1.00	1.00	0.09	1.00	
j	0.99	1.00		0.80	0.99	0.72	1.00	1.00	0.92	0.91	1.00	1.00	0.37	0.87	0.76
k	1.00	0.41		0.96	0.71	0.99	1.00	1.00	0.29	0.55		1.00	0.98	0.81	1.00
l	0.95	0.09		0.87	0.82	0.88	0.92	0.29	1.00	0.46		1.00	0.97	0.11	1.00
n	0.59	0.32	0.84	0.96	0.79	0.99	0.91	0.55	0.46	1.00		1.00	0.98	0.24	1.00
o	1.00					1.00	1.00				1.00		1.00	0.60	1.00
q	1.00	0.92		0.99	0.99	1.00	1.00	1.00	1.00	1.00		1.00	1.00	0.45	1.00
r	0.73	0.96	1.00	0.86	0.98	0.09	0.37	0.98	0.97	0.98	1.00	1.00	1.00	0.90	1.00
s	1.00	0.19	0.93	0.84	0.94	1.00	0.87	0.81	0.11	0.24	0.60	0.45	0.90	1.00	1.00
u	0.55	0.99		0.92	1.00		0.76	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Blank cells indicate an independence probability of less than 0.005.

Unlike the enabling and disabling relationships shown in Tables 6.1 and 6.2, independence is a symmetric relationship, as shown in Table 6.3. In addition, Table 6.3 is less sparse indicating that many phases are typically independent of each other. For instance, it can be seen that *register allocation*(k) is highly independent of most control flow optimizations. *Instruction selection*(s) and *common subexpression elimination*(c) frequently act on the same

code sequences, and so we see a low level of independence between them. Since most of the phases are independent of each other most of the time it is frequently possible to reorder phases without any side-effect. Consequently, many different optimization sequences produce the same code resulting in greater convergence in the DAG and fewer leaf function instances for most functions, as seen in Section 5.4.

6.2 Probabilistic Batch Optimization

The analysis results and observations assembled during our experiments can be further used to improve upon various compiler features. As a case study, we use some of these results to support faster compilations in this section. The VPO compiler applies optimization phases to all functions in one default order. To allow aggressive optimizations, VPO applies many optimization phases in a loop until there are no further program changes produced by any optimization phase. Thus, although VPO can attempt a different number of phases for different functions, the order in which they are attempted still remains the same. Applying optimizations in a loop also means that many optimization phases when attempted are dormant.

We use information about the probability of phases enabling and disabling each other to dynamically select optimizations phases depending on which previous ones were active. The probability of each optimization phase being active by default is used at the start of the optimization process. Using these probabilities as initial values, we dynamically determine which phase should be applied next depending on which phase has the highest probability of being active. After each active optimization phase, we update the probabilities of all other phases depending on the probability that the last phase would enable or disable it. This algorithm is depicted in Figure 6.2. We denote the compiler using this new algorithm of dynamically selecting optimization phases as the *probabilistic batch compiler*.

From Table 6.4 it can be seen that the new probabilistic mode of compilation achieves performance comparable to the old batch mode of compilation and requires less than one-third of the compilation time on average. Although the probabilistic approach reduces the number of attempted phases from over 235, on average, to under 30, the number of active phases is in fact greater in the new approach. Many phases attempted in the old compiler were found by the probabilistic compiler to be disabled and were therefore not attempted. Presently, the probabilistic compiler selects the next phase only on the basis of the probability

```
# p[i] - current probability of phase i being active ;
# e[i][j] - probability of phase j enabling phase i;
# d[i][j] - probability of phase j disabling phase i;
foreach phase i do
    p[i] = e[i][st];     # start phase probabilities (see Table 4) ;
while any p[i] > 0 do
    Select j as the current phase with highest probability of being active;;
    Apply phase j; ;
    if j was active then
        foreach phase i do
            if i != j then
                p[i] += ((1-p[i]) * e[i][j]) - (p[i] * d[i][j]); ;
    p[j] = 0; ;
```

Figure 6.2: **Probabilistic Compilation Algorithm**

of it being active. Our method does not consider the benefits each phase can potentially provide when applied. This is the main reason for the slight degradation in performance, on average, over the old method. Thus, the probabilistic compilation paradigm, even though promising, may be further improved by taking phase benefits into account.

6.3 Evaluating Heuristic Optimization Phase Order Search Algorithms

Exhaustive exploration of the optimization phase order space, although possible in a reasonable amount of time for a large majority of the functions, takes prohibitively long for most large functions to make it suitable for use in typical iterative compilers. Instead, faster heuristic algorithms that scan only a portion of the phase order space are more commonly employed. However, such methods do not evaluate the entire space to provide any guarantees about the quality of the solutions obtained. Commonly used heuristic algorithms to address the phase ordering problem include genetic algorithms [11, 12], hill climbing algorithms [6, 24], as well as random searches of the space [44].

In this section I will present our evaluation of different heuristic search approaches to determine the important characteristics of each algorithm as related to the phase order space. This study is the most detailed evaluation of the performance and cost of different heuristic search methods, and the first to compare their performance with the optimal phase ordering. A major goal of this study is to isolate and evaluate several properties of each studied

Table 6.4: Comparison between the Old Batch and the New Probabilistic Approaches of Compilation

Function	Old Compilation			Prob. Compilation			Prob/Old		
	Attempt Phases	Active Phases	Time	Attempt Phases	Active Phases	Time	Time	Size	Speed
main(i)	253	14	77.55	32	13	35.88	0.463	0.980	0.885
linit(i)	270	15	6.72	44	18	4.29	0.639	1.000	1.000
checkline(i)	270	17	5.67	44	17	3.07	0.541	1.020	1.145
start_inp...(j)	233	16	2.53	32	13	1.20	0.475	1.014	N/A
correct(i)	233	16	11.84	42	18	6.02	0.508	0.996	N/A
main(t)	270	16	4.63	38	19	2.75	0.594	1.000	1.000
parse_swi...(j)	233	14	6.07	34	13	2.93	0.482	1.016	0.972
save_root...(i)	358	29	3.46	64	26	2.11	0.611	1.005	N/A
start_inp...(j)	270	15	1.78	32	14	0.67	0.378	1.007	N/A
start_inp...(j)	233	14	1.53	27	12	0.58	0.381	1.003	N/A
askmode(i)	233	16	2.01	38	17	1.07	0.531	0.997	1.040
skiptoword(i)	270	17	2.44	34	16	1.11	0.454	1.000	1.016
suf_list_...(i)	233	14	1.49	39	16	0.88	0.591	1.003	0.899
start_inp...(j)	231	11	1.02	32	12	0.46	0.448	1.004	1.000
treeoutput(i)	247	16	1.24	38	15	0.73	0.589	1.011	1.769
fft_float(f)	463	28	2.15	48	24	1.22	0.567	1.562	0.974
TeX_skip_...(i)	231	9	1.18	30	10	0.52	0.444	1.004	N/A
treeinit(i)	233	9	1.35	27	11	0.55	0.410	1.023	1.034
pfx_list_...(i)	270	16	1.26	40	16	0.65	0.519	1.013	1.020
main(f)	284	20	1.59	41	18	0.80	0.501	1.003	1.000
flagpr(i)	342	23	1.27	58	22	0.79	0.624	1.018	N/A
makedent(i)	307	22	1.04	47	19	0.47	0.456	1.025	N/A
pat_remove(p)	233	13	0.73	31	14	0.33	0.454	1.018	N/A
sha_trans...(h)	284	18	0.57	45	18	0.38	0.665	1.000	1.000
initckch(i)	344	18	0.86	46	18	0.36	0.422	1.007	1.001
cap_ok(i)	372	25	1.07	59	24	0.57	0.538	1.027	N/A
treeinsert(i)	233	18	0.82	43	19	0.49	0.601	1.020	N/A
expandmode(i)	233	12	0.83	30	12	0.35	0.416	1.012	N/A
read_scan...(j)	233	13	0.81	30	11	0.34	0.421	1.018	N/A
main(p)	233	12	0.90	27	13	0.35	0.387	1.000	1.000
terminit(i)	231	10	0.72	36	14	0.38	0.535	1.012	N/A
LZWReadByte(j)	268	12	0.53	27	11	0.18	0.330	1.014	N/A
pat_insert(p)	233	14	0.61	32	14	0.29	0.485	1.014	1.013
pr_pre_ex...(i)	344	22	0.91	41	16	0.37	0.403	1.014	N/A
main(j)	270	12	0.86	34	15	0.36	0.412	1.007	1.000
main(l)	233	16	0.93	38	18	0.47	0.506	1.013	1.000
dofile(i)	233	10	0.82	27	12	0.32	0.397	1.020	N/A
shellescape(i)	233	15	0.63	33	15	0.31	0.481	1.043	N/A
checkfile(i)	270	15	0.48	37	15	0.23	0.480	1.000	N/A
adpcm_coder(a)	251	13	0.53	32	12	0.31	0.586	2.877	1.197
remaining(204)	228.3	8.6	0.15	26.6	9.4	0.05	0.283	1.020	1.013
average	235.6	9.8	0.75	28.5	10.4	0.35	0.319	1.033	1.021

Old Compilation - original batch compilation, Prob. Compilation - new probabilistic mode of compilation, Attempt Phases - number of attempted phases, Active Phases - number of active phases, Time - compilation time in seconds, Prob/Old - ratio of probabilistic to old compilation for compilation time, code size, and dynamic instruction counts, respectively.

heuristic algorithm, and demonstrate the significance and difficulty in selecting the correct optimization phase sequence length, which is often ignored or kept constant in most previous studies on optimization phase ordering. We also recognize and illustrate the importance of leaf function instances, and show how we can exploit the properties of leaf instances to enhance existing algorithms as well as to construct new search algorithms.

We are able to perform a very thorough study since we had already completely enumerated the phase order spaces of our compiler for hundreds of functions. The experiments in this section also represent the exhaustive phase order space information for each function in the form of a DAG. The DAG then enables much faster evaluation of any search heuristic, since compilation as well as execution can be replaced with a simple table lookup in the DAG to determine the performance of each phase ordering. As a result, the study of the various algorithms can be accomplished very quickly, and it is possible to evaluate various parameters of the algorithms as well as the search space.

Over the past decades researchers have employed various heuristic algorithms to cheaply find effective solutions to the phase ordering problem. However, several issues regarding the relative performance and cost of each algorithm, as well as the effect of changing different algorithm parameters on that algorithm's performance are as yet uncertain and not clearly understood. In this section we will perform a thorough evaluation and comparison of commonly used heuristic methods.

6.3.1 Local Search Techniques

Local search techniques, such as hill climbing and simulated annealing, can only migrate to neighboring points from one iteration to the next during their search for good solutions. Central to these algorithms is the definition of *neighbors* of any point in the space. For this study, we define the neighbors of a sequence to be all those sequences that differ from the base sequence in only one position. Thus, for a compiler with only three optimization phases *a, b* and *c*, the sequence shown in the first column of Table 6.5 will have the sequences listed in the following columns as its neighbors. The position that differs in each neighbor is indicated in bold. For a compiler with m optimization phases, a sequence of length n will have $(m-1)n$ neighbors. Unless the search space is extremely smooth, these local search algorithms have a tendency to get stuck in a local minimum, which are points that are not globally minimum, but have better fitness values than any of their neighbors. For a comprehensive study of

these algorithms it is important to first understand relevant properties of the optimization phase order space. The results from this study are presented in the next section.

Table 6.5: Neighbors in Heuristic Searches

bseq	neighbors							
a	**b**	**c**	a	a	a	a	a	a
b	b	b	**a**	**c**	b	b	b	b
c	c	c	c	c	**a**	**b**	c	c
a	a	a	a	a	a	a	**b**	**c**

Distribution of Local Minima in the Phase Order Space

Earlier studies have attempted to probe the properties of the phase order space [23, 6]. Such studies, however, only looked at a small portion of the space, and ignored important factors affecting the nature and distribution of local minima in phase order spaces. One such factor, commonly ignored, is the optimization sequence length. It is almost impossible to estimate the best sequence length to use due to the ability and tendency of optimization phases to enable other phases. During our experiments, maximum sequence lengths of active phases varied from 3 to 44 over different functions, with considerable variation within the same function itself for different phase orderings. The goal of analyzing the search space, in the context of local search techniques, is to find the properties and distribution of all local minima in each phase order search space. However, there are some difficult hurdles in achieving this goal:

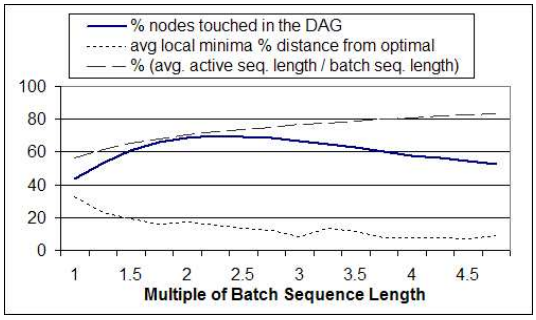
Variable sequence length: Since the best sequence length for each function is unknown, an ideal analysis would require finding the properties of local minima for all possible sequence lengths. This requirement is needed because any sequence of attempted phases of any length defines a point in the search space DAG. Conversely, a single point in the space can be defined by, potentially, infinite number of attempted sequences of different lengths. This is important, since different sequences defining the same point will have different neighbors. This fact implies that some of those sequences may be locally optimum, while others may be not, even though they define the same point in the phase order space. For example, the attempted sequences $\{\mathbf{b} \rightarrow \mathbf{a}\}$, $\{c \rightarrow \mathbf{b} \rightarrow \mathbf{a}\}$, and $\{d \rightarrow \mathbf{b} \rightarrow c \rightarrow \mathbf{a}\}$ all define the same node 4 in the DAG in Figure 5.4 (Note that, the phases **a** and **b**, indicated

in bold, are active, while c and d are dormant). Thus, we can see that it is possible to have sequences of different lengths pointing to the same node. Thus, this ideal goal of finding the local minima for all possible sequence lengths is clearly impossible to achieve.

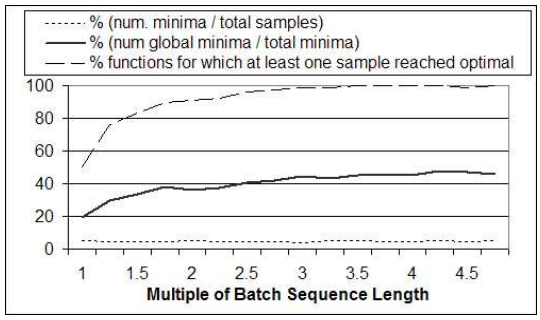
Fixed sequence length: A conceptually simpler approach would be to use some *oracle* to give us the best sequence length to use for each function, and then only analyze the space for this single sequence length. The minimum reasonable length to use, so that all nodes in the DAG can be reached, would be the maximum active sequence length for each function. For an average maximum active sequence length of 16, over all 234 enumerated functions, we would need to evaluate 15^{16} different phase orderings for each function. Evaluation of any phase ordering to determine if that ordering is a local optimum would in turn require us to lookup the performance of that ordering as well as that of its $15 * 16$ neighbors. This, also, is clearly a huge undertaking considering that the maximum active sequence length we encountered during our exhaustive phase order enumeration study was 44.

Due to such issues, in our present experiments, we decided to use *sampling* to probe only a reasonable portion of the phase order search space for some number of different sequence lengths for each function. We use 16 different sequence lengths. The initial length is set to the length of the sequence of active phases applied by the conventional VPO compiler in batch mode. The remaining sequence lengths are successive increments of one-fourth of the initial sequence length used for each function. The larger sequence lengths may be needed to accommodate phases which may be dormant at the point they are attempted. For each set of experiments for each function, we first randomly generate a sequence of the specified length. We then compare the performance of the node that this sequence defines with the performance of all of its neighbors to find if this sequence is a local optimal. This base node is marked as *done*. All later sequences are constrained to define different nodes in the space. As this sampling process progresses it will require an increasing number of attempts to find a sequence corresponding to an unevaluated node in the search space. The process terminates when the average number of attempts to generate a sequence defining a new node exceeds 100.

Figures 6.3(a) and 6.3(b) illustrate the average phase order space properties over all the executed functions that we studied. The plot labeled *% nodes touched in the DAG* from Figure 6.3(a) shows the percentage of nodes that were evaluated for local minimum from amongst all nodes in the space DAG. This number initially increases, reaches a peak, and then



(a) Local Minima Information



(b) Global Minima Information

Figure 6.3: Search Space Properties

drops off. This graph, in effect, shows the nature of typical phase order spaces. Optimization phase order space DAGs typically start out with a small width, reach a maximum around the center of the DAG, and again taper off towards the leaf nodes as more and more function instances generated are detected to be redundant. Smaller attempted sequence lengths in Figure 6.3(a) define points higher up in the DAG, with the nodes defined dropping down in the DAG as the length is increased. The next plot labeled *avg local minima % distance from optimal* in Figure 6.3(a) measures the average difference in performance from optimal over all the samples at each length. As the sequence lengths increased the average performance of the samples gets closer and closer to optimal, until after a certain point the performance remains more or less constant. This is expected, and can be explained from the last plot in Figure 6.3(a), labeled *%(avg.active seq. length / batch seq. length)*, which shows the percentage increase in the average length of active phases as the attempted sequence length is increased. The ability to apply more active phases implies that the function is better optimized, and thus we see a corresponding increase in performance and a smaller percentage of active phases.

The first plot, *%(num minima / total samples)*, in Figure 6.3(b) shows the ratio of sequences reaching local minima to the total sequences probed. This ratio seems to remain more or less constant for different lengths. The small percentage of local minima in the total samples indicates that there are not many local minima in the space. The next plot, *%(num global minima / total minima)*, in this figure shows that the percentage of locally minimum nodes achieving global minima grows with increase in sequence length. This increase is more

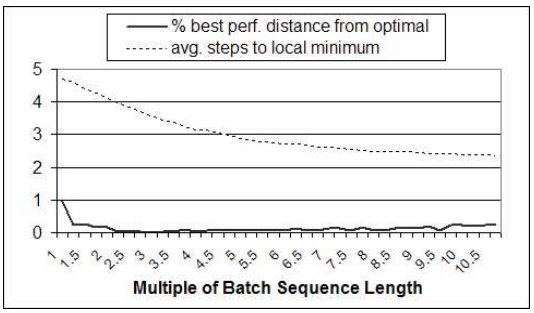
pronounced initially, but subsequently becomes steadier. In the steady state around 45% of local minima display globally optimum performance. This characteristic means that for longer sequence lengths there is a good chance that the local minimum found during local search algorithms will have globally optimal performance. The final plot in Figure 6.3(b), *%(functions for which at least one sample reached optimal)*, presents the percentage of functions for which the probe is able to find optimal in at least one of its samples. This number shows a similar characteristic of continuously increasing with increasing sequence lengths, until it reaches a steady state at close to 100% for larger sequence lengths. Hence, for small multiples of the batch sequence length the local search algorithms should be able to find global minima with a high probability for most of the functions.

Thus, this study illustrates that it is important to find the correct balance between increase in sequence length, performance obtained, and the time required for the search. Although larger sequence lengths tend to perform better, they are also more expensive to evaluate, since they have more neighbors, and evaluation of each neighbor takes longer. It is worthwhile to note that we do not need to increase the sequence lengths indefinitely. After a modest increase in the sequence lengths, as compared to the fixed batch sequence length, we are able to obtain most of the potential benefits of any further increases in sequence lengths.

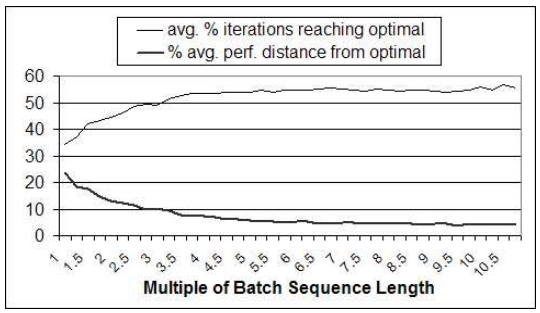
Hill Climbing

In this section we evaluate the performance of the *steepest descent hill climbing* heuristic algorithm for different sequence lengths [6]. The algorithm is initiated by randomly populating a phase sequence of the specified length. The performance of this sequence is evaluated, along with that of all its neighbors. If the best performing neighbor has better performance than the base sequence, then that neighbor is selected as the new base sequence. This process is repeated until a local optimum is reached, i.e., the base sequence performs better than all of its neighbors. For each sequence length, 100 iterations of this algorithm are performed by selecting random starting points in the search space. The sequence lengths were incremented 40 times starting from the length of the active batch sequence, with each increment equal to one-fourth the batch length.

Figures 6.4(a) and 6.4(b) illustrate the results of the hill climbing experiments. The plot marked *% best perf. distance from optimal* in Figure 6.4(a) compares the best solution found by the hill climbing algorithm with optimal, averaged over the 79 executed functions, and



(a) Local Minima Information



(b) Global Minima Information

Figure 6.4: Properties of the Hill Climbing Algorithm

over all 100 iterations for each sequence length. We can see that even for small sequence lengths the algorithm is able to obtain a phase ordering whose best performance is very close to optimal. For lengths greater than 1.5 times the batch sequence length, the algorithm is able to reach optimal in most cases. The plot *avg. steps to local minimum* in Figure 6.4(a) shows that the simple hill climbing algorithm requires very few steps to reach local optimal, and that the average distance to the local optimal decreases with increasing sequence lengths. This decrease in the number of steps is caused by better performance delivered by each typical sequence when the initial sequence length is increased, so that in effect the algorithm starts out with a better initial sequence, and takes fewer steps to the local minimum.

As mentioned earlier, the hill climbing algorithm is iterated 100 times for each sequence length and each function to eliminate the noise caused by the random component of the algorithm. The first plot in Figure 6.4(b), *avg. % iterations reaching optimal*, illustrates that the average number of iterations reaching optimal increases with increase in the sequence length up to a certain limit, after which it remains more or less constant. A related measure *% avg. perf. distance from optimal*, shown in the second plot in Figure 6.4(b), is the average function performance over all the iterations for each sequence length. This measure also shows a marked improvement as the sequence length increases until the average performance peaks at around 4% worse than optimal. These results indicate the significance of selecting a correct sequence length during the algorithm. Larger sequence lengths lead to larger active sequences that result in the initial performance improvement, but increasing the length incessantly gives diminishing returns while making the algorithm more expensive.

Simulated Annealing

Simulated annealing can be defined as a technique to find a good solution to an optimization problem by trying random variations of the current solution. A worse variation is accepted as the new solution with a probability that decreases as the computation proceeds. The slower the cooling schedule, or rate of decrease, the more likely the algorithm is to find an optimal or near-optimal solution [69]. In our implementation, the algorithm proceeds similarly to the hill climbing algorithm by starting from a random initialization point in the phase order space. The sequence length is fixed for each run of the algorithm. During each iteration the performance of the base sequence is evaluated along with that of all its neighbors. Similar to the hill climbing method, if the performance of the best performing neighbor is better than the performance of the base sequence, then that neighbor is selected as the base sequence for the next iteration. However, if the current iteration is not able to find a neighbor performing better than the base sequence, the algorithm can still migrate to the best neighbor based on its current *temperature*. The worse solution is generally accepted with a probability based on the Boltzmann probability distribution:

$$prob = exp(-\frac{\delta f}{T}) \tag{6.1}$$

where, δf is the difference in performance between the current base sequence and the best neighbor, and T is the current temperature. Thus, smaller the degradation and higher the temperature the greater the probability of a worse solution being accepted.

An important component of the simulated annealing algorithm is the *annealing schedule*, which determines the initial temperature and how it is lowered from high to low values. The assignment of a good schedule generally requires physical insight and/or trial and error experiments. In this paper, we attempt to study the effect of different annealing schedules on the performance of a simulated annealing algorithm. For this study, the sequence length is fixed at 1.5 times the batch compiler length of active phases. As seen in the hill climbing experiments, this is the smallest sequence length at which the average performance reaches a steady state that is very close to optimal. We conducted a total of 400 experimental runs by varying the initial temperature and the annealing schedule. The temperature was varied from 0 to 0.95 in steps of 0.5. For each temperature we defined 20 different annealing schedules, which control the temperature in steps from 0.5 to 0.95 per iteration. The results

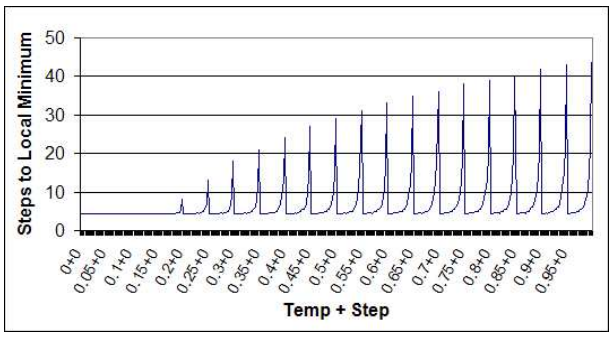


Figure 6.5: Increase in the Number of Steps to Local Minimum with Increases in Initial Temperature and Annealing Schedule Step

for each configuration are averaged over 100 runs to account for noise caused by random initializations.

Our results, shown in Figure 6.5, indicate that for the phase ordering problem, as seen by our compiler, the initial temperature as well as the annealing schedule do not have a significant impact on the performance delivered by the simulated annealing algorithm. The best performance obtained over all the 400 experimental runs is, on average, 0.15% off from optimal, with a standard deviation of 0.13%. Likewise, other measures obtained during our experiments are also consistent across all 400 runs. The average number of iterations achieving optimal performance during each run is 41.06%, with a standard deviation of 0.81%. The average performance for each run is 15.95% worse than optimal, with a deviation of 0.55%. However, as expected, the number of steps to a local minimum during each iteration for each run increases with increase in the initial temperature and the annealing schedule step. As the starting temperature and annealing schedule step are increased, the algorithm accepts more poorly performing solutions before halting. However, this increase in the number of steps to local optimal does not translate into any significant performance improvement for our experiments,

6.3.2 Greedy Algorithm

Greedy algorithms follow the policy of making the locally optimum choice at every step in the hope of finally reaching the global optimum. Such algorithms are commonly used for addressing several optimization problems with huge search spaces. For the phase ordering

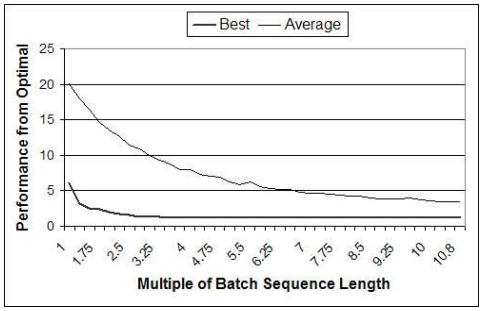


Figure 6.6: Greedy Algorithm Performance

problem, we start off with the empty sequence as the *base* sequence. During each iteration the algorithm creates new sequences by adding each available phase first to the prefix and then as the postfix of the base sequence. Each of these sequences is evaluated to find the best performing sequence in the current iteration, which is consequently selected as the base sequence for the next iteration. If there are multiple sequences obtaining the same best performance, then one of these is selected at random. The algorithm is repeated 100 times in order to reduce the noise that can potentially be caused by this random component in the greedy method. Thus, in our case the algorithm has a bounded cost, as it performs a fixed number of (15+15=30) evaluations in each step, where 15 is the number of available optimizations in our compiler.

Our current implementation of the greedy algorithm is inspired by the approach used by Almagor et al. [6]. Similar to the hill climbing algorithm, the sequence lengths during the greedy algorithm are varied from the active batch sequence length for each function as the initial length to 11 times the batch length, in increments of one-fourth the batch length. To minimize the effect of the random component, the algorithm is repeated 100 times for each sequence length. The best and average performances during these 100 iterations for each sequence length, averaged over all executed functions, are illustrated in Figure 6.6. The plots show a similar pattern to the hill climbing performance graphs. However, it is interesting to note that the best achievable performance during the greedy algorithm is around 1.1% worse than optimal, whereas it is very close to optimal (0.02%) for the hill climbing algorithm. Also, the average performance during the greedy algorithm improves more gradually and continues to improve for larger sequence lengths as compared to hill climbing.

6.3.3 Focusing on Leaf Sequences of Active Phases

As mentioned earlier, leaf function instances are those that cannot be further modified by the application of any additional optimizations phases. These function instances represent leaves in the DAG of the phase order space (e.g. nodes 3, 4, 6, and 8 in Figure 5.4). Sequences of active phases leading to leaf function instances are called *leaf sequences*. Working with only the leaf sequences has the advantage that the heuristic algorithm no longer needs to guess the most appropriate sequence length to minimize the algorithm running time, while at the same time obtaining the best, or at least close to the best possible performance. Since leaf function instances are generated by different lengths of active phase sequences, the length of the leaf sequences is variable. In this section we describe our modifications to existing algorithms, as well as introduce new algorithms that deal with only leaf sequences.

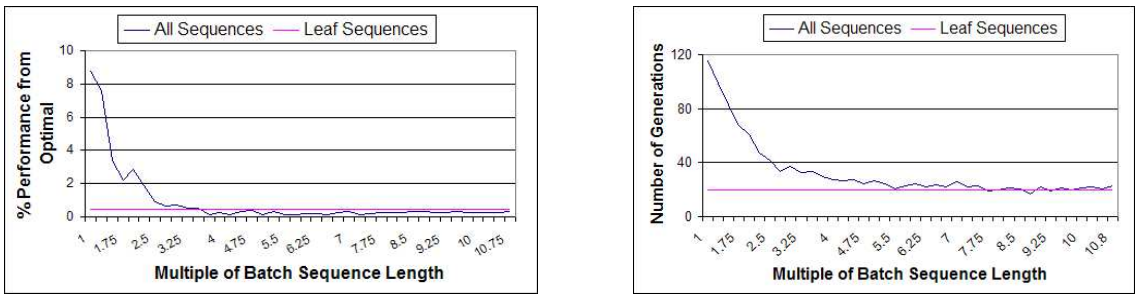
We first motivate the reason for restricting the heuristic searches to only leaf function instances. Figure 5.9 shows the distribution of the dynamic frequency counts as compared to the optimal for all distinct function instances obtained during our exhaustive phase order space evaluation, averaged over all 79 executed functions. From this figure we can see that the performance of the leaf function instances is typically very close to the optimal performance, and that leaf instances comprise a significant portion of optimal function instances with respect to the dynamic frequency counts. This fact is quite intuitive since active optimizations generally improve performance, and very rarely cause a performance degradation. The main drawback of this approach is that the algorithm will not find the optimal phase ordering for any function that does not have an optimal performing leaf instance. However, we have observed that most functions do contain optimal performing leaf instances. For more than 86% of the functions in our benchmark suite there is at least one leaf function instance that achieved optimal dynamic frequency counts. The average best performance for leaf function instances over all executed functions is only 0.42% worse than optimal. Moreover, leaf function instances comprise only 4.38% of the total space of distinct function instances, which is in turn a minuscule portion of the total phase order search space. Thus, restricting the heuristic search to leaf function instances constrains the search to only look at a very small portion of the search space that typically consists of good function instances, and increases the probability of finding a near-optimal solution quickly. In the next few sections we will describe some modifications to existing algorithms, as well

as describe new algorithms that take advantage of the opportunity provided by leaf function instances to find better performance faster.

Genetic Algorithm

Genetic algorithms are adaptive algorithms based on Darwin's theory of evolution [64]. These algorithms have been successfully employed by several researchers to address the phase ordering problem and other related issues in compilers [12, 11, 65, 66]. *Genes* correspond to optimization phases and *chromosomes* correspond to optimization sequences in the genetic algorithm. The set of chromosomes currently under consideration constitutes a *population*. The number of *generations* is how many sets of populations are to be evaluated. Our experiments with genetic algorithms suggests that minor modifications in the configuration of these parameters do not significantly affect the performance delivered by the genetic algorithms. For the current study we have fixed the number of chromosomes in each population at 20. Chromosomes in the first generation are randomly initialized. After evaluating the performance of each chromosome in the population, they are sorted in decreasing order of performance. During crossover, 20% of chromosomes from the poorly performing half of the population are replaced by repeatedly selecting two chromosomes from the better half of the population and replacing the lower half of the first chromosome with the upper half of the second and vice-versa to produce two new chromosomes each time. During mutation we replace a phase with another random phase with a small probability of 5% for chromosomes in the upper half of the population and 10% for the chromosomes in the lower half. The chromosomes replaced during crossover are not mutated.

The only parameter that we have observed to significantly affect the performance of the genetic algorithm is the length of each chromosome. We conducted two different studies with genetic algorithms. In the first study we vary the length of the chromosomes (attempted sequence) starting from the batch sequence length to 11 times the batch sequence length, in steps of one-fourth of the batch length. For the second study we modified the genetic algorithm to only work with leaf sequences of active phases. This approach requires maintaining active leaf sequences of different lengths in the same population. After crossover and mutation it is possible that the new sequences no longer correspond to leaf function instances, and may also contain dormant phases. The modified genetic algorithm handles such sequences by first squeezing out the dormant phases and then extending the sequence, if



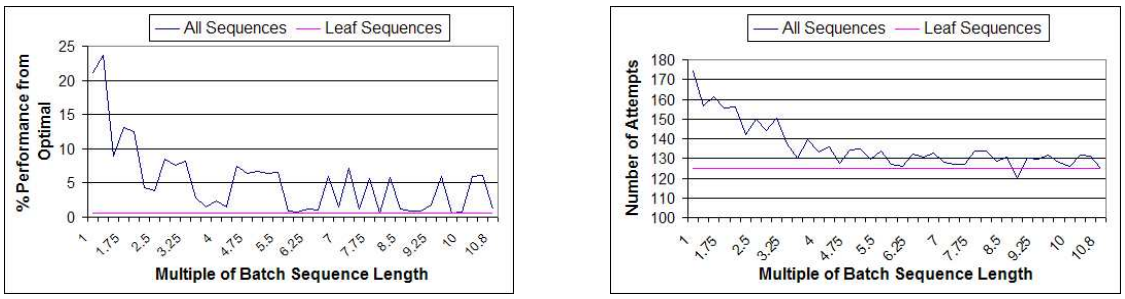
(a) Performance (b) Cost

Figure 6.7: Performance and Cost of Genetic Algorithms

needed, by additional randomly generated phases to get a leaf sequence. Figures 6.7(a) and 6.7(b) shows the performance results, as well as a comparison of the two approaches. Please note that the modified genetic algorithm for leaf instances does not depend on sequence lengths, and is in fact only a single experimental run. To compare the performance and cost of the leaf genetic algorithm with genetic algorithms using fixed sequence lengths, the leaf genetic algorithm measures are represented by single horizontal lines in Figures 6.7(a) and 6.7(b). The number of generations is a measure of the cost of the algorithm. Thus, by concentrating on only the leaf function instances, the genetic algorithm is able to obtain close to the best performance at close to the least cost possible for any sequence length. Interestingly, performance of the genetic algorithm for leaf sequences (0.43%) is very close to the best achievable average leaf performance (0.42%).

Random Search

Random sampling of the search space to find good solutions is an effective technique for search spaces that are typically discrete and sparse, and when the relationships between the various space parameters are not clearly understood. Examples are the search spaces to address the phase ordering problem. In this study we have attempted to evaluate random sampling, again by performing two different sets of experiments similar to the genetic algorithm experiments in the previous section. For the first set, randomly constructed phase sequences of different lengths are evaluated until 100 consecutive sequences fail to show an improvement over the current best. The second set of experiments is similar, but only considers leaf sequences or leaf function instances.



(a) Performance (b) Cost

Figure 6.8: Performance and Cost of Random Search Algorithms

Figures 6.8(a) and 6.8(b) show the performance benefits as well as the cost for all our random search experiments. As noted in the previous section, the performance and cost of the random algorithm for leaf instances are represented by horizontal lines in Figures 6.8(a) and 6.8(b). It is interesting to note that random searches are also able to achieve performance close to the optimal for each function in a few number of attempts. Since our algorithm configuration mandates the best performance to be held steady for 100 consecutive sequences, we see that the cost of our algorithm is always above 100 attempts. We again notice that leaf sequences consistently obtain good performance for the random search algorithm as well. In fact, for our current configuration, random search algorithm concentrating on only the leaf sequences is able to cheaply outperform the best achievable by any other random search algorithm for any sequence length.

N-Lookahead Algorithm

This algorithm scans N levels down the search space DAG from the current location to select the phase that leads to the best subsequence of phases to apply. The critical parameter is the number of levels to scan. For a N lookahead algorithm we have to evaluate 15^N different optimization phases to select each phase in the base sequence. This process can be very expensive, especially for larger values of the lookahead N. Thus, in order for this approach to be feasible we need to study if small values of the lookahead N can achieve near optimal performance for most of the functions.

For the current set of experiments we have constrained the values of N to be either 1, 2, or 3 levels of lookahead. Due to the exponential nature of the phase order search space, we

believe that any further increase in the lookahead value will make this method too expensive in comparison with other heuristic approaches. Table 6.6 shows the average performance difference from optimal for the three levels of lookahead over all the executed functions in our set. As expected, the performance improves as the levels of lookahead are increased. However, even after using three levels of lookahead the performance is far from optimal. This illustrates the ragged nature of typical phase order search spaces, where it is difficult to predict the final performance of a phase sequence by only looking at a few number of phases further down from the current location.

Table 6.6: Perf. of N-Lookahead Algorithm

	Lookahead		
	1	2	3
% Performance	22.90	14.64	5.35

6.3.4 Summary of Results

We have a number of interesting conclusions from our detailed study: (1) Analysis of the phase order search space indicates that the space is highly discrete and very sparse. (2) The phase order space typically has a few local and global minima. More importantly, the sequence length of attempted phases defines the percentage of local and global minima in the search space. Larger sequence lengths increase the probability of finding a global minima, but can also increase the search time to find a good solution. Thus, it is important to find the correct sequence lengths to balance algorithm cost, and its ability to reach better performing solutions faster. (3) On comparing the performance and cost of different heuristic algorithms we find that simple techniques, such as local hill climbing allowed to run over multiple iterations, can often outperform more complex techniques such as genetic algorithms and lookahead schemes. The added complexity of simulated annealing, as compared to hill climbing, is found to not significantly affect the performance of the algorithm. Random searches and greedy search algorithms achieve decent performance, but not as good as the other heuristic approaches for the amount of effort expended. The unpredictable nature of phase interactions is responsible for the mediocre performance of the N-lookahead heuristic algorithm. (4) Due to the inherent difficulty in determining the ideal sequence length to use

during any heuristic method, and the high probability of obtaining near-optimal performance from leaf function instances, we modified existing algorithms to concentrate on only leaf sequences and demonstrated that for many algorithms leaf sequences can deliver performance close to the best, and often times even better than that obtained by excessive increases in sequence lengths for the same algorithms. Moreover, this can be achieved at a fraction of the running cost of the original algorithm since the space of leaf function instances is only 4.38% of the total space of all function instances. (5) Interestingly, most of the heuristic algorithms we evaluated are able to achieve performance close to the best phase ordering performance in acceptable running times for all functions. Thus, in conclusion we find that differences in performance delivered by different heuristic approaches are not that significant when compared to the optimal phase ordering performance. Selection of the correct sequence length is important for algorithms that depend on it, but can be safely bypassed without any significant performance loss wherever possible by concentrating on leaf sequences.

6.4 Miscellaneous Topics

In this section I will describe some other interesting phase order search space properties and analysis results.

6.4.1 Statically Predicting Optimization Phase Order Space Size

Out of the 244 functions from the MiBench benchmark suite we were able to completely enumerate the phase order space for 234 of these functions. Even though we were unable to exhaustively enumerate the space for the remaining 10 functions, we still had to spend considerable time and resources for their partial search space evaluation before realizing that their function complexity was greater that the capability of our current space pruning techniques to contain the space growth. For such cases, it would be very helpful if we could a priori determine the complexity of the function and estimate the size of the search space statically. This determination is, however, very hard to achieve. The growth in the phase order space is dependent on various factors such as the number of conditional and unconditional transfers of control, loops and loop nesting depths, as well as the number of instructions and instruction mix in the function.

In this study we have attempted to calibrate the function complexity based on static

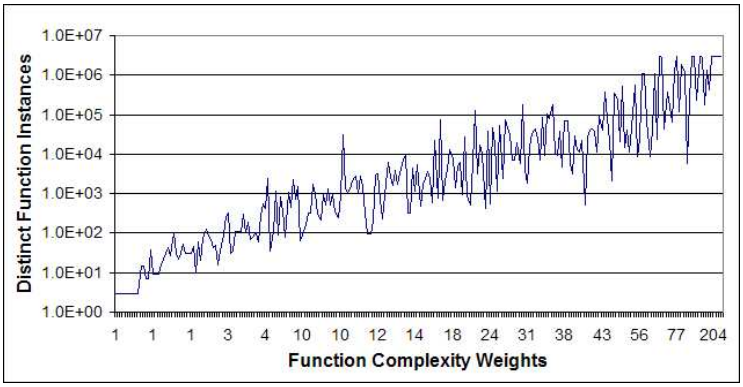


Figure 6.9: Function Complexity Distribution

function features such as branches, loops, and number of instructions. All transfers of control are assigned a unit value. Loops at the outermost level are assigned a weight of 5 units. All successive loop nesting levels are weighted two times the weight of the preceding loop level. Functions with the same weight are sorted based on the number of instructions in the unoptimized function instance. The 10 unevaluated functions are assumed to have 3,000,000 distinct function instances, which are more than the number of instances for any function in our set of evaluated functions.

Figure 6.9 shows the distribution of the number of distinct function instances for each function as compared to its assigned complexity weight. Figure 6.9 shows a marked increase in the number of distinct function instances with increase in the assigned complexity weights. A significant oscillation of values in the graph reconfirms the fact that it is difficult to accurately predict the number of distinct function instances based on static function features. It is, however, interesting to note that out of the 10 unevaluated functions, five are detected to have the highest complexity, with eight of them within the top 13 complexity values. Thus, a static complexity measure can often aid the complier in effectively prioritizing the functions for exhaustively evaluating the phase order space.

6.4.2 Redundancy Found by Each Space Pruning Technique

As described in Section 5.3 we employ three techniques to exploit redundancy in the optimization phase order space: detecting dormant phases, detecting identical function instances, and detecting equivalent function instances. The *attempted search space* for

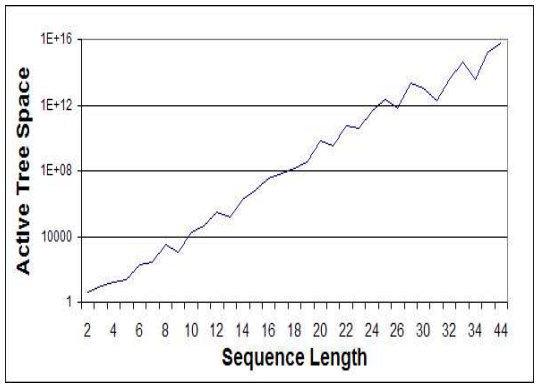


Figure 6.10: Active Search Space for Different Sequence Lengths

any function in our compiler with 15 optimization phases is 15^n, where n is the maximum active sequence length for that function. The length of the longest active sequence for various functions is listed in Tables 5.2 and 5.3. As explained earlier, phases that are unsuccessful in changing the function when applied are called *dormant* phases. Eliminating the dormant phases from the attempted space results in the *active search space*. Thus, the active search space only consists of phases that are *active*, i.e. successful in changing the program representation when attempted. Figure 6.10 shows the resulting average number of function instances in the active space for each maximum active sequence length. The fraction of the active space to the attempted space, particularly for functions with larger sequence lengths, is so small that we could not easily plot the ratios. This shows the drastic reduction in attempted search space that is typically obtained by detecting and eliminating dormant phases from the search space.

The remaining pruning techniques find even more redundancy in the active search space. These pruning techniques detect identical and equivalent function instances, which causes different branches of the active search space tree to merge together. Thus, the active search space tree is converted into a DAG. Figure 6.11 shows the average ratio of the number of distinct function instances to the number of nodes in the active search space tree. Thus, this figure illustrates the redundancy eliminated by detecting identical and equivalent function instances. The fraction of the tree of function instances that is distinct and represented as a DAG decreases as the active sequence length increases. This property of exponential increase in the amount of redundancy detected by our pruning techniques as sequence lengths are

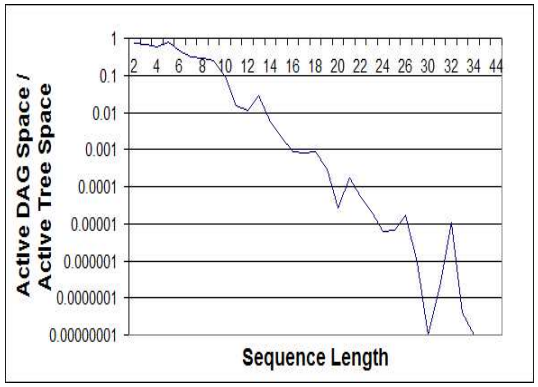
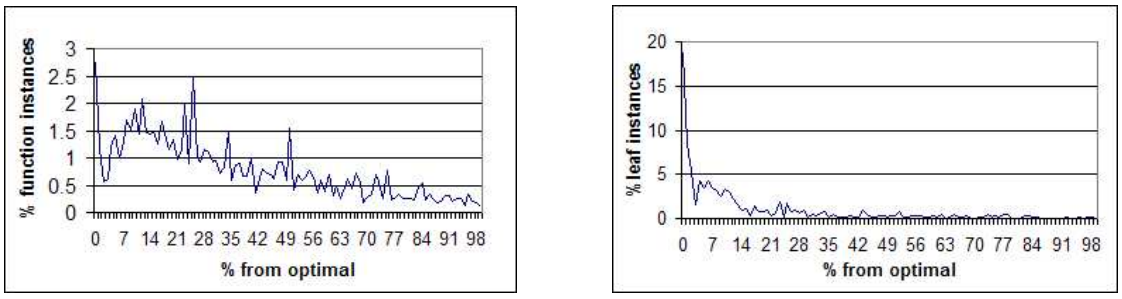


Figure 6.11: Ratio of Distinct Function Instances in Active Space



(a) (b)

Figure 6.12: Distribution of Performance Compared to Optimal

increased is critical for exhaustively exploring the search spaces for larger functions.

6.4.3 Performance Comparison with Optimal

Figure 6.12(a) shows the distribution of the dynamic frequency count performances of all function instances as compared to the optimal performance. Figure 6.12(b) illustrates a similar distribution over only the leaf function instances. For both of these graphs, the numbers have been averaged over all the 79 executed functions in our benchmark suite. Function instances that are over 100% worse than optimal are not plotted. On average, 22.27% of total function instances, and 4.70% of leaf instances fall in this category.

Figure 6.12(a) rates the performances of all function instances, even the un-optimized and partially optimized instances. As a result very few instances compare well with optimal. We expect leaf instances to perform better since these are fully optimized function instances

that cannot be improved upon by any additional optimizations. Accordingly, over 18% of leaf instances on average yield optimal performance. Also, a significant number of leaf function instances are seen to perform very close to optimal. This is an important result, which can be used to seed heuristic algorithms, such as genetic algorithms, with leaf function instances to induce them to find better phase orderings faster.

6.4.4 Optimization Phase Repetition

Figure 6.13 illustrates the maximum number of times each phase is active during the exhaustive phase order evaluation over all studied functions. Functions with the same maximum sequence length are grouped together, and the maximum phase repetition number of the entire group is plotted. The functions in the figure are sorted on the basis of the maximum sequence length for that function. The optimization phases in Figure 6.13 are labeled by the *codes* assigned to each phase in Table 3.1.

Common subexpression elimination(c) and *instruction selection*(s) are the phases that are typically active most often in each sequence. These phases perform the work of cleaning up the code after many other optimizations, and hence are frequently enabled by other phases. For example, *instruction selection*(s) is required to be performed before *register allocation*(k) so that candidate load and store instructions can contain the addresses of arguments or local scalars. However, after allocating locals to registers, *register allocation*(k) creates many additional opportunities for *instruction selection*(s) to combine instructions. For functions with loops, *loop transformations*(l) may also be active several times due to freeing up of registers, or suitable changes to the instruction patterns by other intertwining optimizations. Most of the branch optimizations, such as *branch chaining*(b), *reverse branches*(r), *block reordering*(i), and *remove useless jumps*(u), are not typically enabled by any other optimizations, and so are active at most once or twice during each optimization sequence. For this study, we restricted *loop unrolling* to be active at most once for each sequence.

6.5 Concluding Remarks

In this chapter I explained how we can analyze the phase order space over several different functions, and then employ observations about optimization phase interactions and phase

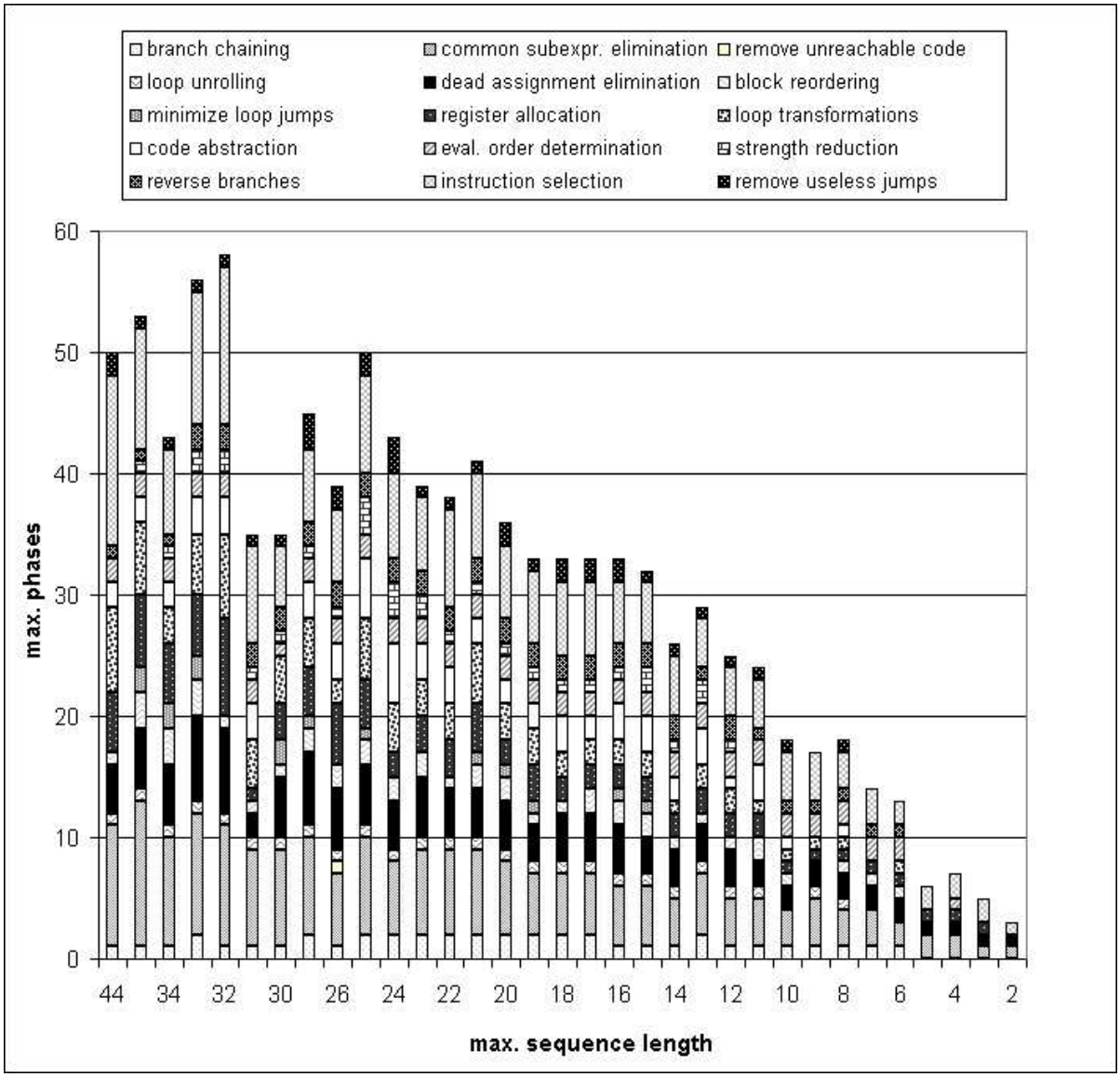


Figure 6.13: Repetition Rate of Active Phases

order characteristics to improve other compilation tasks. Investigation of the phase order space can reveal interesting optimization phase interactions, such as the enabling, disabling, and independence interactions presented in this chapter. I further showed how we used the enabling and disabling phase interactions to improve our default compiler to produce comparably performing code in less than one-third of the compilation time.

In another study we compared the cost and performance of commonly used heuristic

phase order search algorithms. We showed the importance of selecting an appropriate optimization phase sequence length to properly balance algorithm cost and performance. We discovered that focusing on leaf instances can also prove beneficial for several heuristic algorithms. Based of such observations, we further proposed and evaluated techniques to improve current heuristic searches.

CHAPTER 7

Future Work

The ultimate goal of this research is to improve performance of compiler generated code by practically finding near-optimal optimization phase orderings on a per-function basis. Even though a lot of the infrastructure to achieve exhaustive phase order space evaluation is already in place, there is still room for several other enhancements. All my current work has focused on *pruning* the phase order space. There is still a lot of work needed to actually *reduce* the phase order space. This can be achieved by changing the implementation of some compiler analysis and optimizations, so that false dependences due to incomplete analysis are no longer present. Phases that can be proved to never interact (independent phases) can then be merged into one pass for the purposes of phase ordering. This will allow exhaustive searches of functions with large search spaces, which we have currently ignored. There is also some potential to develop more aggressive pruning techniques. This follows from our observation that many function instances detected to be distinct by our pruning techniques perform equivalently. The main difficulty here is to verifiably show that such instances will also interact identically with future optimization phases. In the future we also plan to parallelize the search algorithm by simultaneously starting several threads exploring different parts of the phase order search space.

We believe that there is still a lot of unexplored analysis that needs to be done to better understand, or even to simply use the phase interaction information. Such analysis can then be used to improve both the conventional compilers, as well as approximate heuristic search approaches. Presently the only feedback we get from each optimization phase was whether it was active or dormant. We do not keep track of the number and type of actual changes for which each phase is responsible. Keeping track of this information would be very useful to get more accurate phase interaction information. Probabilistic statistics on

the effect of phase interactions on application performance can be utilized in our probabilistic batch compilers to improve program performance at the same time as reducing compilation time. The enabling/disabling relationships between phases could be used for faster genetic algorithm searches [17]. Machine learning techniques have been shown to be fruitful to focus probabilistic algorithms on high performance areas of the phase order space based on correlating function features, and previous phase order behavior. The pre-existing knowledge can be greatly enhanced by exhaustive searches to further improve probabilistic searches.

CHAPTER 8

Conclusions

The phase ordering problem has been an important and long-standing problem in compiler optimizations. The large number of optimization phases present in current compilers, along with the phase reordering flexibility provided by many compilers creates a huge space of distinct phase orderings for every function or application. As a result, exhaustive evaluation of the complete phase order space to find the best phase ordering for every function was always considered to be impossible. At the same time, the observation of different phase orderings producing different relative performances for different functions made it important to effectively and consistently resolve the phase ordering problem in all cases.

A major result of my dissertation is the realization that exhaustive phase order space evaluation to find the optimal phase ordering (for all the phases in the VPO compiler) is practically possible for most functions. Over the course of my research I developed various techniques and algorithms to make this evaluation feasible. The re-interpretation of the phase ordering problem to concentrate efforts on generating all possible distinct *function instances* that could be produced by different phase orderings, instead of evaluating all possible phase orderings themselves was instrumental in making the phase ordering problem more manageable. This re-interpretation at once makes the problem much more manageable, since we had already shown that many different phase orderings can potentially produce the same code.

Practically enumerating all distinct function instances for each function required novel applications of known search algorithms, and a new representation of the phase order space in the form of a Directed Acyclic Graph (DAG). We also developed aggressive pruning techniques to exploit redundancy in the phase order space by detecting dormant phases, and identical/equivalent function instances. We also designed more efficient and accurate

estimation methods by categorizing function instances based on their basic-block control flows. We verified the accuracy of our performance estimations by showing strong correlation between our performance estimations and cycle counts obtained from a cycle-accurate simulator. All these techniques allowed us to find near-optimal optimization phase orderings in a reasonable amount of time for 234 out of the 244 functions that we studied.

Exhaustive evaluation of such a large number of functions provided us with a huge data space and good understanding of optimization phase interactions and phase ordering properties. We have also described several other potential applications of using this phase interaction information. We used the enabling and disabling phase interactions to improve our conventional compiler by producing comparable performing code in less than one-third the compilation time. We also conducted a survey of existing heuristic algorithms for finding effective phase orderings, by comparing their costs and performances with each other, as well as with the optimal phase ordering performance. A better understanding of the phase ordering characteristics allowed us to demonstrate the importance of selecting an appropriate sequence length to balance the cost of the heuristic algorithm with the quality of the solution produced. We also showed how we can exploit leaf function instances during heuristic algorithms to reach better performing phase orderings earlier in the search.

In summary, I believe that we have made significant strides in solving one of the most well known problems in compiler optimization that has been deemed intractable for over 40 years. I further believe that our approach for efficient and exhaustive evaluation of optimization phase order space has opened a new area of compiler research.

APPENDIX A

Function Statistics

Table A.1: Phase Order Statistics Over All Functions in MiBench

Function	Inst	Block	Br	Lp	Fn_inst	Len	CF	Leaf	Batch/optimal	
									Size	Perf.
start_in...(j)	1371	88	69	2	120777	25	70	894	1.014	N/A
correct(i)	1294	244	106	5	1348154	25	663	7231	1.042	N/A
main(t)	1275	126	110	6	2882021	29	389	15164	1.163	1.000
parse_sw...(j)	1228	198	144	1	180762	20	53	2057	1.004	1.067
start_in...(j)	1009	72	55	1	39352	21	18	336	1.025	N/A
start_in...(j)	971	82	67	1	63458	21	30	388	1.017	N/A
askmode(i)	942	112	84	3	232453	24	108	475	1.079	1.084
skiptowo...(i)	901	173	144	3	439994	22	103	2834	1.015	1.061
start_in...(j)	795	63	50	1	8521	16	45	80	1.027	1.017
TeX_skip...(i)	704	87	77	2	5734	15	30	180	1.022	N/A
treeinit(i)	666	77	59	0	8940	15	22	240	1.024	1.000
pfx_list...(i)	640	77	59	2	1269638	44	136	4660	1.084	1.043
main(f)	624	50	35	5	2789903	33	122	4214	1.210	1.075
makedent(i)	555	57	47	2	1063697	33	70	5325	1.026	N/A
pat_remo...(p)	552	78	62	4	1151047	24	59	1669	1.006	N/A
sha_tran...(h)	541	33	25	6	548812	32	98	5262	1.638	1.096
initckch(i)	536	65	48	2	1075278	32	32	4988	1.028	1.000
treeinse...(i)	510	64	48	3	368810	26	75	1000	1.027	N/A
expandmo...(i)	493	56	41	2	23530	22	15	372	1.006	N/A
main(p)	483	39	26	1	14510	15	10	178	1.013	1.077
read_sca...(j)	480	59	52	2	44489	18	57	791	1.012	N/A
terminit(i)	476	50	33	1	3072	15	32	56	1.012	N/A
LZWReadB...(j)	472	44	33	2	39434	22	19	189	1.007	N/A
pat_inse...(p)	469	51	41	4	1088108	25	71	3021	1.000	1.000
main(j)	465	40	28	1	25495	21	12	134	1.000	1.056
main(l)	464	66	51	4	1896446	25	920	5364	1.097	1.009
dofile(i)	436	50	38	0	5700	16	12	136	1.000	N/A
shellesc...(i)	420	71	53	5	244264	21	161	3054	1.022	N/A
checkfile(i)	416	45	36	5	154348	24	234	2345	1.234	N/A
adpcm_co...(a)	385	51	35	1	28013	23	24	230	1.009	1.018
dijkstra(d)	354	30	22	3	92973	22	18	1356	1.066	1.000
givehelp(i)	347	15	10	1	584	11	9	31	1.011	N/A
usage(j)	344	3	1	0	34	9	1	3	1.000	N/A
TeX_math...(i)	344	58	49	2	69841	30	50	147	1.037	N/A
casecmp(i)	342	41	34	2	366006	31	37	1804	1.090	N/A
GetCode(j)	339	14	11	1	74531	22	20	117	1.047	N/A
show_char(i)	333	64	49	1	45581	17	152	812	1.296	N/A
good(i)	313	37	29	1	87206	22	32	370	1.000	1.000
bmhi_init(s)	309	30	22	4	11640	22	11	188	1.022	N/A
adpcm_de...(a)	306	41	28	1	44422	29	61	369	1.000	N/A
chk_aff(i)	304	34	30	1	179431	21	160	2434	1.082	1.001
cpTag(t)	303	54	40	0	522	11	9	16	1.070	1.016
update_f...(i)	294	29	22	1	6874	17	14	64	1.022	N/A
processC...(t)	289	41	31	0	1804	13	10	38	1.034	N/A

continued on next page

Table A.1: *continued...*

Function	Inst	Block	Br	Lp	Fn_inst	Len	CF	Leaf	Batch/optimal	
									Size	Perf.
toutent(i)	286	35	29	1	7158	17	17	19	1.032	N/A
combinec...(i)	284	26	19	1	45350	19	75	108	1.062	N/A
stringch...(i)	284	41	30	2	81318	24	40	1184	1.000	N/A
makeposs...(i)	280	45	33	1	70368	24	119	498	1.141	1.000
xgets(i)	273	47	37	1	37960	19	103	284	1.000	1.000
preload_...(j)	268	29	22	2	4890	16	10	29	1.000	N/A
combine_...(i)	263	20	15	0	2352	14	8	26	1.042	N/A
missings...(i)	262	35	28	3	23477	26	30	513	1.023	1.040
pr_suf_e...(i)	258	30	23	1	19522	20	59	184	1.000	N/A
missingl...(i)	252	40	31	3	11524	16	40	180	1.040	1.129
get_inte...(j)	249	20	17	1	16900	20	10	86	1.052	N/A
bmha_init(s)	248	31	22	3	44868	25	83	835	1.242	N/A
chk_suf(i)	243	25	21	1	75628	21	29	2835	1.050	1.008
findfile...(i)	241	39	29	3	15014	18	50	266	1.000	N/A
read_qua...(j)	239	25	21	2	9171	16	28	296	1.000	N/A
acoversb(i)	239	42	36	0	10450	16	16	70	1.000	N/A
load_int...(j)	235	19	13	2	6992	17	8	20	1.017	N/A
save_cap(i)	226	21	16	1	123843	22	44	1476	1.246	N/A
TeX_math...(i)	225	29	24	0	568	12	8	18	1.029	N/A
compound...(i)	222	37	30	1	78429	20	49	448	1.000	1.111
main(b)	220	22	15	2	182246	23	84	508	1.026	1.083
get_word...(j)	220	11	7	1	3288	18	4	92	1.114	N/A
TeX_LR_b...(i)	220	28	23	0	424	10	8	18	1.000	N/A
read_col...(j)	218	20	16	2	2440	17	14	40	1.604	N/A
addvhead...(i)	217	13	10	0	1631	13	6	28	1.038	N/A
dumpmode(i)	214	31	25	0	1160	13	7	40	1.061	N/A
skipover...(i)	212	39	30	1	105353	29	110	413	1.047	1.077
set_samp...(j)	209	38	33	2	21818	21	51	253	1.038	N/A
entdump(i)	205	25	17	1	3256	14	12	54	1.000	N/A
bmha_sea...(s)	201	29	24	3	530754	40	409	2638	1.000	N/A
lookhard...(i)	197	28	20	1	18011	18	41	288	1.107	N/A
tinsert(i)	196	20	13	1	7458	16	10	40	1.017	N/A
lookup(i)	195	27	22	2	37396	20	38	114	1.000	1.077
whatcap(i)	195	42	34	3	20527	19	44	520	1.077	N/A
bmh_init(s)	194	21	15	3	5940	21	11	129	1.037	N/A
wronglet...(i)	194	32	25	2	22065	17	25	430	1.017	1.150
ichartos...(i)	186	35	26	3	40524	21	40	304	1.038	1.000
bmhi_sea...(s)	184	29	24	3	243309	34	312	1359	1.000	N/A
bmh_sear...(s)	181	29	24	3	356817	33	492	1766	1.000	N/A
TeX_LR_c...(i)	180	21	15	0	3732	14	8	60	1.000	N/A
TeX_math...(i)	179	23	19	0	967	11	6	36	1.000	N/A
show_line(i)	178	31	24	3	12647	21	23	236	1.065	N/A
getline(i)	176	37	27	2	40982	23	123	122	1.013	N/A
main(s)	175	19	12	3	30980	23	10	163	1.086	1.000
main(d)	175	21	15	3	9206	20	22	85	1.000	1.043
main(q)	174	19	14	2	38759	23	121	160	1.298	1.000
get_8bit...(j)	171	8	5	1	2800	16	4	47	1.059	N/A
treelook...(i)	167	28	23	2	67507	17	65	1992	1.022	1.000
get_scal...(j)	166	11	7	1	1784	13	4	42	1.000	N/A
insertto...(i)	163	26	19	4	28980	22	64	378	1.017	N/A
insertR(p)	161	20	15	0	2462	14	6	22	1.000	1.005
get_16bi...(j)	158	8	5	1	1040	13	4	31	1.000	N/A
set_quan...(j)	156	30	25	2	11065	16	49	38	1.017	N/A
preload_...(j)	156	14	9	1	1120	13	8	13	1.023	N/A
subsetdu...(i)	156	33	24	3	2166	17	63	28	1.562	N/A
sha_final(h)	155	7	4	0	2472	13	3	68	1.000	1.000
setdump(i)	155	23	16	1	6364	19	83	92	1.118	N/A
usage(i)	153	3	1	0	16	5	1	3	1.000	N/A
select_f...(j)	149	25	21	0	510	10	10	16	1.000	1.000
TeX_skip...(i)	147	26	21	0	871	12	17	32	1.061	N/A
byte_rev...(h)	146	8	5	1	2715	19	13	54	2.500	1.004
get_24bi...(j)	145	8	5	1	2168	15	4	44	1.069	N/A

continued on next page

Table A.1: *continued...*

Function	Inst	Block	Br	Lp	Fn_inst	Len	CF	Leaf	Batch/optimal	
									Size	Perf.
get_word...(j)	144	11	7	1	3816	17	4	92	1.061	N/A
read_tex...(j)	141	21	16	2	52861	20	34	422	1.071	N/A
main(a)	140	16	10	1	1676	16	8	12	1.000	1.000
strtoich...(i)	140	25	18	1	10721	19	17	109	1.056	1.000
read_sca...(j)	139	27	22	1	46248	23	47	207	1.021	N/A
ntbl_bit...(b)	138	3	1	0	48	7	1	8	1.000	1.000
read_pbm...(j)	134	27	21	2	4182	15	18	60	1.020	1.067
bitcount(b)	133	3	1	0	44	8	1	7	1.000	1.000
get_8bit...(j)	131	8	5	1	1352	14	4	40	1.000	N/A
strsearch(s)	128	23	17	2	32550	17	48	972	1.025	1.015
insert(i)	128	18	14	1	6053	17	22	63	1.025	N/A
get_scal...(j)	126	11	7	1	1592	12	4	37	1.000	N/A
enqueue(d)	124	15	10	1	488	13	4	12	1.029	1.002
get_text...(j)	123	8	5	1	1068	13	4	7	1.000	N/A
read_col...(j)	123	11	7	1	2802	17	12	46	1.412	N/A
sha_upda...(h)	118	11	7	1	5990	18	50	49	1.462	1.001
get_pixe...(j)	117	8	5	1	1292	13	4	33	1.087	N/A
transpos...(i)	117	14	10	1	5310	16	19	44	1.032	1.045
read_rle...(j)	116	18	13	1	13356	16	7	39	1.000	N/A
expand_s...(i)	111	16	11	1	761	13	13	25	1.549	N/A
pat_sear...(p)	110	20	14	1	5052	15	33	98	1.000	1.006
compress...(t)	106	8	5	1	30885	32	3	142	1.083	N/A
get_24bi...(j)	104	8	5	1	868	15	4	20	1.000	N/A
init_sea...(s)	103	13	9	2	1430	15	11	30	1.034	1.004
main(h)	101	16	11	1	22476	20	129	320	1.000	1.071
expand_p...(i)	97	11	7	1	959	13	13	25	1.575	N/A
compress...(t)	94	8	5	1	240	10	3	11	1.000	1.000
ins_root...(i)	94	11	8	1	9746	15	20	101	1.057	N/A
keymatch(j)	92	20	16	1	26939	19	20	126	1.062	N/A
extralet...(i)	91	17	13	1	4278	15	28	56	1.031	1.057
trydict(i)	88	10	6	0	815	11	6	33	1.100	1.167
sha_init(h)	87	3	1	0	68	8	1	5	1.000	1.000
Index_to...(f)	87	19	12	0	234	10	6	12	1.000	N/A
toutword(i)	85	11	7	1	712	13	10	15	1.000	N/A
AR_btbl_...(b)	83	3	1	0	87	9	1	10	1.000	1.000
compress...(t)	82	8	5	1	345	14	3	10	1.000	N/A
ReadColo...(j)	79	8	5	1	533	14	6	8	1.120	N/A
get_text...(j)	79	8	5	1	1040	13	4	7	1.000	N/A
entryhas...(i)	79	13	11	0	318	8	6	16	1.000	N/A
forcevhe...(i)	79	13	9	0	1538	12	9	90	1.067	N/A
usage(t)	78	8	5	1	1324	15	4	11	1.000	N/A
dequeue(d)	76	6	3	0	102	7	3	14	1.042	1.000
bstr_i(b)	70	12	9	1	4437	16	11	45	1.000	N/A
GetDataB...(j)	70	9	5	0	90	7	2	5	1.000	N/A
get_8bit...(j)	70	8	5	1	504	10	4	9	1.000	N/A
strtosic...(i)	70	6	3	0	78	8	2	6	1.000	N/A
get_memo...(j)	69	3	1	0	129	8	1	12	1.053	N/A
checkcmap(t)	69	15	12	1	2106	16	24	24	1.048	N/A
BW_btbl_...(b)	68	3	1	0	72	8	1	7	1.000	1.000
pat_count(p)	68	6	4	0	430	11	5	4	1.000	N/A
cpTags(t)	68	8	5	1	943	16	5	8	1.000	1.000
copyout(i)	68	17	12	1	696	12	13	20	1.038	1.068
TeX_strn...(i)	67	11	8	0	2190	12	8	88	1.048	N/A
DoExtens...(j)	66	3	1	0	20	6	1	6	1.095	N/A
ichartos...(i)	66	6	3	0	70	8	2	6	1.000	N/A
printich...(i)	65	7	4	0	561	12	3	40	1.000	N/A
setbit(b)	64	17	9	0	736	11	5	16	1.033	N/A
InitLZWC...(j)	64	3	1	0	60	9	1	4	1.000	N/A
print_pa...(d)	63	6	3	0	188	13	2	8	1.062	1.054
bfopen(b)	62	9	7	0	1062	11	10	28	1.000	N/A
TeX_skip...(i)	62	12	9	1	330	11	6	19	1.235	N/A
TeX_open...(i)	62	12	9	1	330	11	6	19	1.235	N/A

continued on next page

Table A.1: *continued...*

Function	Inst	Block	Br	Lp	Fn_inst	Len	CF	Leaf	Batch/optimal	
									Size	Perf.
onstop(i)	61	3	1	0	10	4	1	2	1.000	N/A
sha_print(h)	60	3	1	0	45	8	1	7	1.000	1.091
get_raw_...(j)	60	6	3	0	105	10	1	10	1.000	1.000
bfwrite(b)	59	6	3	0	309	9	2	10	1.000	N/A
NumberOf...(f)	59	11	7	1	3235	15	10	80	1.000	1.289
bfread(b)	58	8	4	0	273	10	2	10	1.000	N/A
sha_stre...(h)	55	8	5	1	210	13	4	4	1.050	1.092
ins_cap(i)	55	11	8	1	7088	14	20	51	1.056	N/A
read_byte(j)	52	6	3	0	112	9	2	3	1.000	N/A
jinit_re...(j)	52	3	1	0	30	6	1	2	1.000	N/A
ReadByte(j)	52	6	3	0	112	9	2	3	1.000	N/A
jinit_re...(j)	52	3	1	0	30	6	1	2	1.000	N/A
read_byt...(j)	52	6	3	0	112	9	2	3	1.000	N/A
jinit_re...(j)	52	3	1	0	30	6	1	2	1.000	N/A
issubset(i)	52	7	5	0	1178	10	5	93	1.000	N/A
myfree(i)	52	10	8	0	440	12	8	21	1.062	N/A
ReInitLZW(j)	51	3	1	0	52	9	1	4	1.000	N/A
__bswap_...(p)	51	3	1	0	30	7	1	7	1.000	N/A
treeload(i)	51	8	5	1	296	11	4	4	1.000	N/A
line_size(i)	50	10	7	1	2905	12	9	36	1.059	N/A
read_non...(j)	48	8	5	1	1011	12	3	6	1.000	N/A
btbl_bit...(b)	47	6	3	0	322	9	2	12	1.000	N/A
bit_shif...(b)	47	10	7	1	200	8	9	3	1.000	1.000
wrongcap...(i)	47	6	3	0	36	6	3	4	1.118	1.100
flagout(i)	47	6	3	0	30	6	1	6	1.000	N/A
jinit_re...(j)	45	3	1	0	22	6	1	2	1.000	1.000
ntbl_bit...(b)	43	6	3	0	272	10	2	20	1.200	1.397
forcelc(i)	43	8	5	1	1739	21	13	83	1.000	N/A
done(i)	42	9	5	0	112	9	2	9	1.000	N/A
getbit(b)	41	11	5	0	34	8	1	3	1.056	N/A
pbm_getc(j)	41	11	7	1	96	9	7	4	1.077	1.000
text_getc(j)	41	11	7	1	96	9	7	4	1.077	N/A
upcase(i)	41	8	5	1	327	12	3	7	1.000	1.000
lowcase(i)	41	8	5	1	327	12	3	7	1.000	N/A
chupcase(i)	41	3	1	0	30	7	1	1	1.000	N/A
flipbit(b)	38	9	4	0	61	8	1	4	1.000	N/A
compare(q)	37	11	7	0	81	7	6	6	1.083	1.259
bit_count(b)	36	9	5	1	169	13	4	6	1.250	1.014
CheckPoi...(f)	35	6	3	0	93	7	3	16	1.100	1.500
combinea...(i)	33	3	1	0	106	8	1	12	1.000	N/A
IsPowerO...(f)	30	9	7	0	378	10	6	24	1.000	1.000
pdictcmp(i)	26	3	1	0	27	5	1	4	1.000	N/A
__bswap_...(p)	25	3	1	0	42	7	1	4	1.000	N/A
tbldump(i)	25	8	5	1	92	11	8	7	1.818	N/A
alloc_bi...(b)	24	3	1	0	30	6	1	3	1.000	N/A
tryveryh...(i)	23	3	1	0	22	6	1	3	1.167	1.167
SkipData...(j)	19	8	5	1	66	6	3	2	1.000	N/A
posscmp(i)	18	3	1	0	16	5	1	2	1.333	N/A
bfclose(b)	16	3	1	0	9	4	1	1	1.000	N/A
read_std...(j)	16	3	1	0	9	4	1	2	1.000	N/A
write_st...(j)	16	3	1	0	9	4	1	2	1.000	1.000
bit(p)	15	3	1	0	37	7	1	4	1.000	1.000
bhmi_cle...(s)	14	3	1	0	7	3	1	1	1.000	N/A
qcount(d)	12	3	1	0	7	3	1	1	1.000	1.000
putch(i)	11	3	1	0	15	5	1	2	1.500	N/A
mymalloc(i)	11	3	1	0	15	5	1	2	1.500	N/A
stop(i)	9	3	1	0	3	2	1	1	1.000	N/A
finish_i...(j)	5	3	1	0	3	2	1	1	1.000	N/A
finish_i...(j)	5	3	1	0	3	2	1	1	1.000	N/A
finish_i...(j)	5	3	1	0	3	2	1	1	1.000	1.000
finish_i...(j)	5	3	1	0	3	2	1	1	1.000	N/A

continued on next page

Table A.1: *continued...*

Function	Inst	Block	Br	Lp	Fn_inst	Len	CF	Leaf	Batch/optimal	
									Size	Perf.
erase(i)	5	3	1	0	3	2	1	1	1.000	N/A
move(i)	5	3	1	0	3	2	1	1	1.000	N/A
inverse(i)	5	3	1	0	3	2	1	1	1.000	N/A
normal(i)	5	3	1	0	3	2	1	1	1.000	N/A
backup(i)	5	3	1	0	3	2	1	1	1.000	N/A
average (234)	196.2	23.7	17.3	1.1	89946.7	14.7	36.2	458.3	1.065	1.048

Table A.1: (Function - function name followed by benchmark indicator [(a)-adpcm, (b)-bitcount, (d)-dijkstra, (f)-fft, (h)-sha, (i)-ispell, (j)-jpeg, (l)-blowfish, (q)-qsort, (p)-patricia, (t)-tiff, (s)-stringsearch]), (Inst - number of instructions in unoptimized function), (Block - number of basic blocks in unoptimized function), (Br - number of conditional and unconditional transfers of control), (Lp - number of loops), (Fn_inst - number of distinct control-flow instances), (Len - largest active optimization phase sequence length), (CF - number of distinct control flows), (Leaf - Number of leaf function instances), (% Batch/optimal - % code size and dynamic performance ratio of the batch Vs. the best phase ordering).

REFERENCES

[1] Jean E. Sammet. *Programming Languages: History and Fundamentals*. Prentice-Hall, Inc., 1969.

[2] John Backus. The history of fortran i, ii, and iii. *SIGPLAN Notices*, 13(8):165–180, 1978.

[3] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. J. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The fortran automatic coding system. In *Proceedings of the Western Joint Computer Conference*, pages 188–198, February 1957.

[4] 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. ACM Press, 1988.

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

[6] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd 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, New York, NY, USA, 2004. ACM Press.

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

[8] 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*, March 26-29 2006.

[9] Steven 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.

[10] 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. ACM Press, 1990.

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

[12] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle 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. ACM Press, 2003.

[13] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 204–215. IEEE Computer Society, 2003.

[14] Marc Auslander and Martin Hopkins. An overview of the pl.8 compiler. In *Proceedings of the ACM SIGPLAN Notices '82 Symposium on Compiler Construction*, pages 22–31, Boston, June 1982.

[15] Shlomit S. Pinter. Register allocation with instruction scheduling: A new approach. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 248–257, 1993.

[16] Cindy Norris and Lori L. Pollock. An experimental study of several cooperative register allocation and instruction scheduling strategies. In *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 169–179, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[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*, June 2004.

[18] Prasad A. Kulkarni, Stephen R. Hines, David B. Whalley, Jason D. Hiser, Jack W. Davidson, and Douglas L. Jones. Fast and efficient searches for effective optimization-phase sequences. *ACM Transactions on Architecture and Code Optimization*, 2(2):165–198, 2005.

[19] Prasad Kulkarni, David Whalley, Gary Tyson, and Jack Davidson. Practical exhaustive optimization phase order exploration and evaluation. submitted in the ACM Transactions on Programming Languages and Systems, October 2006.

[20] Prasad Kulkarni, David Whalley, Gary Tyson, and Jack 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, New York, NY, USA, 2006. ACM Press.

[21] Prasad Kulkarni, David Whalley, Gary Tyson, and Jack Davidson. Evaluating heuristic optimization phase order search algorithms. In *to appear in the IEEE/ACM International Symposium on Code Generation and Optimization*, March 2007.

[22] Prasad A. Kulkarni. Performance driven optimization tuning in vista. Master's thesis, Florida State University, July 2003.

[23] F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, , and E. Rohou. Iterative compilation in a non-linear optimisation space. Proc. Workshop on Profile and Feedback Directed Compilation.Organized in conjuction with PACT'98, 1998.

[24] T. Kisuki, P. Knijnenburg, , and M.F.P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proc. PACT*, pages 237–246, 2000.

[25] P. M. W. Knijnenburg, T. Kisuki, K. Gallivan, and M. F. P. O'Boyle. The effect of cache models on iterative compilation for combined tiling and unrolling: Research articles. *Concurr. Comput. : Pract. Exper.*, 16(2-3):247–270, 2004.

[26] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).

[27] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *Proceedings of the 11th international conference on Supercomputing*, pages 340–347. ACM Press, 1997.

[28] Frigo, Matteo, Johnson, and Steven G. FFTW: An adaptive software architecture for the FFT. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.

[29] M. Frigo. A Fast Fourier Transform Compiler. In *PLDI'99 — Conference on Programming Language Design and Implementation*, Atlanta, GA, 1999.

[30] M. Frigo and S. G. Johnson. The fastest fourier transform in the west. Technical Report MIT/LCS/TR-728, 1997.

[31] Jakob Ostergaard. Optimqr - a software package to create near-optimal solvers for sparse systems of linear equations. http://ostenfeld.dk/ jakob/OptimQR.

[32] See homepage for a complete list of the people involved. Signal processing algorithms implementation research for adaptable libraries. http://www.ece.cmu.edu/ spiral.

[33] See homepage for a complete list of the people involved. Tune - mathematical models, transformations and systems support for menory-friendly programming. http://www.cs.unc.edu/Research/TUNE.

[34] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 279–290, New York, NY, USA, 1995. ACM Press.

[35] Steve Carr. Combining optimization for cache and instruction-level parallelism. In *PACT '96: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, page 238, Washington, DC, USA, 1996. IEEE Computer Society.

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

[37] George E. P. Box, William G. Hunter, and J. Stuart Hunter. *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*. John Wiley & Sons, 1 edition, June 1978. isbn:0471093157.

[38] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizatons. Proc. 2nd Workshop on Feedback Directed Optimization, 1999.

[39] R.P.J. Pinkers, P.M.W. Knijnenburg, M. Haneda, and H.A.G. Wijsholt. Analysis of compiler options using orthogonal arrays. In *Proceedings of CPC*, pages 137–148, 2004.

[40] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Optimizing general purpose compiler optimization. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 180–188, New York, NY, USA, 2005. ACM Press.

[41] E. Granston and A. Holler. Automatic recommendation of compiler options. In *Proceedings 4th Feedback Directed Optimization Workshop*, December 2001.

[42] Zhelong Pan and Rudolf 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, Washington, DC, USA, 2006. IEEE Computer Society.

[43] Sid-Ahmed-Ali Touati and Denis Barthou. On the decidability of phase ordering problem in optimizing compilation. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 147–156, New York, NY, USA, 2006. ACM Press.

[44] 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. IEEE Computer Society.

[45] Min Zhao, Bruce Childers, and Mary Lou 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.

[46] Min Zhao, Bruce R. Childers, and Mary Lou 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.

[47] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, 1993.

[48] Shun Long and Grigori Fursin. A heuristic search algorithm based on unified transformation framework. In *7th workshop on High Performance Scientific and Engineering Computing*, Norway, 2005. IEEE Computer Society.

[49] Henry Massalin. Superoptimizer: a look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 122–126. IEEE Computer Society Press, 1987.

[50] Torbjrn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pages 341–352. ACM Press, 1992.

[51] P.M.W. Knijnenburg, T. Kisuki, K. Gallivan, and M.F.P. O'Boyle. The effect of cache models on iterative compilation for combined tiling and unrolling. In *Proc. FDDO-3*, pages 31–40, 2000.

[52] K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Acme: Adaptive compilation made efficient. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 69–78, June 15-17 2005.

[53] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate static estimators for program optimization. *SIGPLAN Not.*, 29(6):85–96, 1994.

[54] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE International Symposium on Microarchitecture*, pages 274–286, Washington, DC, USA, 1996. IEEE Computer Society.

[55] F. Irigoin and R. Triolet. Supernode partitioning. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 319–329, New York, USA, 1988. ACM Press.

[56] Guang R. Gao, R. Olsen, Vivek Sarkar, and Radhika Thekkath. Collective loop fusion for array contraction. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 281–295, London, UK, 1993. Springer-Verlag.

[57] M. E. Benitez and J. W. Davidson. The advantages of machine-dependent global optimization. In *Proceedings of the 1994 International Conference on Programming Languages and Architectures*, pages 105–124, March 1994.

[58] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[59] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[60] W. Zhao, B. Cai, D. Whalley, M. Bailey, R. van Engelen, X. Yuan, J. Hiser, J. Davidson, K. Gallivan, and D. Jones. Vista: A system for interactive code improvement. In *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 155–164. ACM, June 2002.

[61] Prasad Kulkarni, Wankang Zhao, Steve Hines, David Whalley, , Xin Yuan, Robert van Engelen, Kyle Gallivan, Jason Hiser, Jack Davidson, Baosheng Cai, Mark Bailey, Hwashin Moon, Kyunghwan Cho, Yunheung Paek, and Douglas Jones. Vista: Vpo interactive system for tuning applications. volume 5, pages 819–863, November 2006.

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

[63] John H. Holland. *Adaptation in natural and artificial systems.* University of Michigan Press, 1975.

[64] Melanie Mitchell. *An Introduction to Genetic Algorithms.* Cambridge, Mass. MIT Press, 1996.

[65] A. Nisbet. Genetic algorithm optimized parallelization. Workshop on Profile and Feedback Directed Compilation, 1998.

[66] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 77–90. ACM Press, 2003.

[67] W. Peterson and D. Brown. Cyclic codes for error detection. In *Proceedings of the IRE*, volume 49, pages 228–235, January 1961.

[68] Eric W. Weisstein. Correlation coefficient. from MathWorld - A Wolfram Web Resource, May 2006. http://mathworld.wolfram.com/CorrelationCoefficient.html.

[69] Paul E. Black. Simulated annealing. Dictionary of Algorithms and Data Structures adopted by the U.S. National Institute of Standards and Technology, December 2004. http://www.nist.gov/dads/HTML/simulatedAnnealing.html.

BIOGRAPHICAL SKETCH

Prasad A. Kulkarni

The author was born on October 13^{th}, 1979. He received his Bachelor of Engineering (B.E.) in Computer Engineering from Maharashtra Institute on Technology, Poona University, India in August, 2001. He received his Master of Science degree in Computer Science from Florida State University in August, 2003, where his thesis dealt with research on the *phase ordering problem* in compiler optimizations, and enhancing the VISTA interactive compilation environment. After graduation with his M.S. degree he continued at Florida State University to pursue a Ph.D. in Computer Science. His research interests include compilers, computer architecture, and embedded systems.