RICE UNIVERSITY

On Accelerating the Searches for Compilation Sequences in
an Adaptive Compiler

by

Yi Guo

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Keith D. Cooper, co-Chair
Professor of Computer Science

Devika Subramanian, co-Chair
Professor of Computer Science

Kartik Mohahiani
Assistant Professor of ECE

HOUSTON, TEXAS

FEB 2007
INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI Microform 1441823
Copyright 2007 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.
Abstract

On Accelerating the Searches for Compilation Sequences in an Adaptive Compiler

by

Yi Guo

Recent research show that adaptive compiler can produce consistent improvement over a traditional fixed-sequence compiler by conducting feedback-directed searches for good compilation sequences for specific programs, machines and performance objectives. However, such improvement is usually achieved at very high search cost. This thesis proposes two approaches to accelerate the searches for a good compilation sequence in an adaptive compiler. First, a local search algorithm, Greedy Neighbor Exploration algorithm (GNE), is proposed. It uses optimistic greedy construction and cleanup procedures to generate a richer set of meaningful variations by randomized insertion and removal of transformations. Experimental results on a range of standard benchmark suites show that GNE finds better compilation sequences in less than a quarter of the evaluations required by current search algorithms, such as genetic and hill climbing algorithms. Second, code normalization techniques are developed to hash programs and detect equivalent code. This can avoid unnecessary runs of programs.
Acknowledgements

First, I would like to thank my advisor Professor Keith D. Cooper. Without his support, the thesis would not have been possible. His vision of the future compilers made the research of adaptive compiler a hot topic both in and outside of Rice.

I am grateful to my thesis co-director Professor Devika Subramanian for helping me in all aspects of this research.

I thank Professor Kartik Mohanram for serving on my thesis committee and the valuable feedback to improve my work.

I thank all the current and former members in the Massively Scalar Compiler Project at Rice. This work is not possible without any one of you. Especially, I thank Dr. Alexander Grosul for the guidance when I was new to the area. I thank Dr. Timothy Harvey who is always a very helpful resource for my questions. I thank Steven Reeves who provided SPARC backend support to this work. I also enjoyed the helpful discussion with Jeffery Sandoval. Again, thank all of you for the support, inspiration and feedback.

Last but not least, I am indebted to my parents in China for their unconditional love and support.
Contents

1 Introduction ........................................... 1
   1.1 Adaptive Compiler .................................. 3
   1.2 Organization of the Thesis .......................... 6

2 Related Work ......................................... 7
   2.1 Transformation Ordering ............................ 8
   2.2 Parameter Tuning .................................... 12

3 Background ........................................... 14
   3.1 Experimental Framework ............................. 14
      3.1.1 Platform ....................................... 14
      3.1.2 Transformation Set ............................ 15
   3.2 Optimization Space ................................. 16
   3.3 Review of Current Search Algorithms .............. 18
      3.3.1 Genetic Algorithms ............................ 18
      3.3.2 Hill Climbing Algorithms ...................... 19
      3.3.3 Greedy Construction .......................... 19
      3.3.4 Discussion ................................... 20

4 GNE: Greedy Neighbor Exploration .................... 22
   4.1 Outline ............................................ 22
   4.2 Neighbor Exploration ............................... 24
      4.2.1 Cleanup ....................................... 25
      4.2.2 Optimistic Greedy Construction .............. 27
   4.3 Discussion ......................................... 29

5 Code Normalization and Redundancy Removal ............ 32
   5.1 Code Normalization ................................. 32
   5.2 Redundancy Removal ............................... 37
   5.3 Experiment ........................................ 38

6 Experimental Results .................................. 40
   6.1 Genetic Algorithms and Hill Climbing .............. 41
   6.2 Comparing Searching Performance .................. 46
   6.3 Conclusion ........................................ 50

7 Conclusion and Future Work .......................... 52
List of Figures

1.1 Traditional Compiler Design ........................................... 2
1.2 Prototype of an Adaptive Compiler ................................. 4

4.1 Algorithm Outline ..................................................... 23
4.2 Neighbor Exploration .................................................. 29

5.1 Simple Code Normalization - Code A ......................... 33
5.2 Simple Code Normalization - Code B ......................... 34
5.3 Code A and Code B’s SSA form and their Normalized SSA form ... 35
5.4 $\phi$-Node Encoding .................................................. 36
5.5 Hashing and Redundancy Removal ................................. 38

6.1 Genetic and Hill Climbing Algorithm for Benchmark fmin, zeroin, svd ......................................................... 43
6.2 Genetic and Hill Climbing Algorithm for Benchmark spline, si, fehl 44
6.3 Genetic and Hill Climbing Algorithm for Benchmark bitcnts, sha 45
6.4 Genetic and Hill Climbing Algorithm (Averaged over all Benchmarks) 46
6.5 Search Algorithms Comparison for Benchmark .................. 48
6.6 Search Algorithms Comparison for Benchmark - cont. .......... 49
6.7 Average Search Algorithms Performance ......................... 49
**List of Tables**

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The Transformation Set of Our Adaptive Compiler</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>Characteristic of the Optimization Space</td>
<td>17</td>
</tr>
<tr>
<td>5.1</td>
<td>Percentage of Equivalent Code Detected</td>
<td>39</td>
</tr>
<tr>
<td>6.1</td>
<td>Benchmarks used in the Experiments</td>
<td>41</td>
</tr>
<tr>
<td>6.2</td>
<td>Speedup over built-in sequence of the best sequence found by GNE, GA and HC after 100, 200, 400 and 800 trials</td>
<td>50</td>
</tr>
<tr>
<td>6.3</td>
<td>Number of trials required to achieve 70%, 80% and 90% of the potential improvement on SPARC</td>
<td>50</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The fundamental structure of a compiler has not been changed since the FORTRAN Automatic Coding System was designed by IBM in 1957. As shown in Figure 1.1, a typical optimizing compiler is composed of a series of passes in three stages: the front-end transforms the source code to intermediate representation; the middle-end applies various optimizations on the intermediate representation; the back-end produces code on the specific target. Each stage consists of several transformation passes which are applied, in sequential order, to optimize the outcome code.

Before the adaptive compiler, the compilation sequence of a traditional compiler was designed and fixed ahead-of-time. The compiler used the same strategy for all input programs. However, designing a good compilation sequence that is good for all programs is impractical. Schielke (9) and Grosul (16) both showed that program-specific optimization sequences outperform fixed sequences and that the best sequences show wide variation from program to program.

One of the main reasons is the transformation selection problem. Some transformations tend to be special-purpose optimizations that only benefit a certain class of
Figure 1.1: Traditional Compiler Design

programs (8). Their inclusion in a fixed compilation sequence can hurt other programs. Some transformations, for example, different variations of value numbering, may overlap and the best selection depends on the input program, target architecture and performance objective. Some transformations, for example, loop peeling, may be applied multiple times during the compilation to achieve maximum performance; however, the best number of applications may vary for different input code in different systems.

The problem is even more complicated when transformations can interact with each other. For example, loop unrolling may create opportunities for better instruction scheduling and operator strength reduction; however, it may also increase register pressure. Whether loop unrolling is beneficial at one point in the sequence not only depends on the input code and target architecture, but also depends on the follow up transformations such as instruction scheduling, operator strength reduction, dead code elimination and register allocation etc.
Allowing the user to control a small number of flags, such as \texttt{-O2}, \texttt{-O3} or \texttt{-inline} etc in \texttt{gcc} provides some flexibility in controlling the compiler’s behavior and therefore improve optimization for some input programs. However, there are two main shortcomings to this method. First, specifying a set of good flags requires understanding of the compiler optimization as well as the properties of the input programs and the target architecture. Different performance objectives, such as optimizing for speed or code size, need different optimizing strategies. It is difficult for a normal user to pick the best set of flags for such a complex compiler system. Second, these flags do not prove enough control over optimizations to be effective (27).

1.1 Adaptive Compiler

In contrast to the traditional fixed-sequence compiler, an adaptive compiler automatically customizes the compilation sequence to each given input program, target architecture and performance objective. Our prototype of an adaptive compiler is shown in Figure 1.2.

The key part of the adaptive compiler is the compile-evaluate-feedback loop. It compiles the source code using the compilation sequence given by the steering algorithm. Next, it evaluates the output code produced and the measurement is returned as feedback to the steering algorithm. Depending on the user’s objective, the measurement can be either retrieved by static analysis or through the run of the program. For example, if optimizing for code size, there is no need to run the program. If opti-
mizing for code speed, then running the compiled program or estimating its runtime is necessary. The steering algorithm receives feedback from the evaluation part and tunes the compilation sequences accordingly. The steering algorithm is essentially a search algorithm that explores the optimization space composed of all possible compilation sequences.

There are two main challenges in designing an adaptive compiler:

- **Modular transformation** Since the compilation transformations are applied in fixed order in a traditional compiler, some transformations may assume some code annotations that are made by previous transformations. Sometimes, two transformations may be mixed into a single pass of the program. However, in adaptive compilers, to allow maximum flexibility in tuning the transformation order, each transformation should be designed to be as modular as possible so
that it will produce valid output code for any valid input code.

- **Search Algorithm** Finding the best compilation sequence for a given input program is basically a search process. The search space consists of all possible compilation sequences. The size of the space is exponential in the number of transformations and the length of the sequence. For example, given the 16 transformations in our pool, the size of the space of all sequences of length $m$ is $16^m$. In practice, the length of a good sequence is usually greater 10, which leads to a very large space. Besides, acquiring the measurement of a single sequence will go through the compile-evaluate-feedback loop, and the time it takes is not negligible, especially when a run of the program is necessary. The huge search space and the time it takes to evaluate a single sequence make it impossible to evaluate all sequences in the space. Therefore, one big challenge is to design an incomplete search algorithm that delivers a good compilation sequence by sampling a small portion of the space.

Current search algorithms, such as hill climbing and genetic algorithms, usually require several thousands of trials to find a high quality compilation sequence in an adaptive compiler (16). The high search cost is the primary cause that makes adaptive compilation hard to implement for large real world applications. Two main approaches have been studied in the literature to improve the performance of adaptive compilers: a better search algorithm and redundancy removal during the search. This thesis examines both approaches and delivers better solutions.
This thesis presents new methods that find better compilation sequences in less than a quarter of the evaluations required by current search algorithms. First, a new search algorithm called Greedy Neighbor Exploration (GNE) is presented. GNE uses optimistic greedy construction and cleanup procedures to generate a richer set of meaningful variations by randomized insertion and removal of transformations. Second, an SSA-based code normalization technique is developed to hash programs and detect equivalent code to avoid unnecessary runs of programs during the search.

1.2 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 reviews related work in the field of adaptive compilation. Chapter 3 describes the background of this work, which includes the definition of terms, description of the experimental framework, optimization space overview, and a performance evaluation of current search algorithms. Chapter 4 describes GNE, the greedy neighbor exploration algorithm. Chapter 5 describes the code normalization technique that is used in the system to detect equivalent code to avoid unnecessary evaluations. Chapter 6 experimentally shows the search performance of current search algorithms and compares them to GNE. Chapter 7 concludes the thesis.
Chapter 2

Related Work

It has been long known that a single, fixed compilation strategy is unlikely to produce good code for all inputs programs (16; 31; 19). The studies of this area can be divided into several categories from different perspectives.

From the methodology perspective, both theoretical and empirical approaches have been taken. The goal of theoretical approach is to build a model that can predict the effect of the transformation on an input program, and use the model to suggest the best compilation strategy given an input program (30; 31).

The empirical approach finds its best compilation strategy by compiling and/or running the program multiple times and keeps the best one found (16; 19; 1; 26; 29; 15). Such compilers are also called iterative compilers. They usually have a search algorithm to explore the optimization space. Most researchers, including us, take the empirical approach, since building a precise model for compiler and input programs is very difficult. The disadvantage of the empirical approach is its long compilation time. Most researches in this area, including this thesis, focus on reducing the optimization space.
From the prospective of the range of the input programs to the compiler, some studies focus on a general-purpose compiler (15; 30; 16; 19; 1; 26) while others focus on domain-specific compilers that are designed to optimize a particular class of input programs (31; 29; 17). Our adaptive compiler is a general-purpose compiler.

From the prospective on how the compiler is tuned to provide different compilation strategies, some focus on reordering the transformations (30; 16; 19; 1) and others focus on tuning the compiler parameters for either an individual transformation (29; 17) or the whole compiler (15; 26). The former approach may require major modification to the existing compiler; however, it may deliver optimizing opportunities that cannot be utilized by changing the parameter settings. Ideally, we can utilize the power of both transformation reordering and parameter tuning, but the research remains open primary because neither problem has been well solved.

In the rest of this chapter, we review the papers both in transformation ordering and parameter tuning.

2.1 Transformation Ordering

Witfield and Soffa take a theoretical approach (30). They characterize the transformations in terms of code properties that must be present in order for a transformation to have an effect or properties that could prevent the effect. They analytically derive a table of enabling and disabling interactions between transformations. This table is useful when designing a built-in sequence for general programs, but does
not address the customized compilation problem for adaptive compilers that may use different compilation sequences for different programs.

Zhao et al. built on these ideas and described a model-based framework to predict the impact of optimizations given the resource and code context (31). The predictive framework is used to select the most beneficial optimization for loop nests. Zhao’s work is domain-specific and focuses on the data cache performance for embedded systems.

Schielske’s 1999 paper is the first use of a genetic algorithm to find a good compilation transformation ordering (9). He showed improvement in both code size and execution speed. The framework of the GAs used in the paper is similar to those used in current research adaptive compilers such as VISTA (19). Each gene represents a transformation, and mutation and 1-point crossover are applied to the sequences of the population. Schielke ran this GA for 100 generations with a population of 100 to find a good sequence, which totals 10000 trials. Schielke showed that by using GAs to reordering the compilation transformations, we can get produce smaller and/or faster code, but at a very high cost.

Several techniques have been proposed to improve the genetic algorithms’ searching performance. Some work removes the redundancy in evaluating different sequences, others are looking for a better searching algorithm.

L. Almagor did some unpublished study to improve the search performance of Schielke’s genetic algorithms by optimizing its parameter settings and using the
“mutation-until-unique” technique which forces the sequence to mutate if it has already been seen before. In Grosul’s Ph.D. thesis (16), he characterized the properties of a subspace of size $5^{10}$ and compared several search algorithms including randomized probing, the improved genetic algorithms, hill climbing algorithms, and greedy construction in both the $5^{10}$ subspace and the $16^{10}$ space. He found that in the larger space, hill climbing algorithm can find effective compilation sequences within approximately 500 trials, whose quality are comparable to those GA finds within 5000 trials.

This thesis uses the same GA framework and adopts optimized genetic and hill-climbing algorithms’ parameter settings used in Grosul’s thesis, except the length of representations when comparing the current search algorithms to GNE. The idea of a greedy construction algorithm (GC) in Grosul’s thesis is fundamental to GNE, although GC was reported to be inferior to other algorithms for the reasons described in Chapter 4. Comparing to Grosul’s work, GNE explores a larger space without the sequence length limitation. Thus those sequences considered good in Grosul’s space may be no longer good in our space.

Kulkarni et al. also worked on the transformation ordering and selection problem (19). Their framework called VISTA is based on VPO (Very Portable Optimizer), a compiler that performs optimization on a single low-level representation called RTL (register transfer lists), which is a counterpart to our ILOC. They proposed techniques to speed up searches for compilation sequence in genetic algorithms: detecting and
removing redundant trials of equivalent programs and prohibiting certain dormant or disabled transformations. I implemented their techniques to detect identical and equivalent code and found a 30% savings in the number of trials. This number is much less than 87% which was reported in their paper. This can be attributed to two reasons: First, with Almagor’s, “mutation-until-unique” heuristic, our GA can evaluate many more unique sequences than before, given the same population size and the number of generations. Second, the properties of the optimization spaces are different: our transformation set can produce a greater number of distinct versions of intermediate code on our benchmarks. In another paper (20), they exhaustively explored the optimization space of most of their benchmark programs. However, this is infeasible in our system because of some aggressive code-changing transformations, especially loop peeling, algebraic re-association (4) and strength reduction (10). Every invocation of the transformation is likely to change the code and there is no convergence guarantee.

Statistical and machine learning techniques have been used to improve the performance of searching algorithms. Agakov et al. selected a set of benchmarks and learned an offline model for each benchmark (1). When given a new program, the model of the benchmark that is most similar to the new program is used to focus the search space. This is an attempt to combine modeling with the empirical approach. The problems with this method are how to select a typical benchmark for offline training and how much is the cost if a wrong model is mistakenly selected.
Our future work includes investigating how statistical and machine learning models can improve the performance of our search algorithms.

2.2 Parameter Tuning

Some researches on adaptive compilation use a different mechanism to tune the compiler. Some find a good parameter settings when invoking a compiler; Some focus on selecting the best parameters for a specific transformation.

Triantafyllis et al. demonstrated the promise of using multiple compilation configurations in a practical general-purpose compiler (26). Their framework Optimization-Space-Exploration (OSE) explores a small set of predetermined optimization options and selects the best one for a given input program posteriori. To save on compilation speed, OSE uses static analysis instead of actual running of the program.

Granston and Holler developed a tool called Dr. Options that attempts to recommend the best options to achieve high performance when using Hewlett-Packard’s PA-RISC optimizing compilers (15). Practical use of the tool has shown it to be a very helpful consultant, and in a few cases, it even beat the experts in the performance of the code generated.

Kisuki et al. used various search algorithms to find good optimization settings, such as tile sizes, unroll factors, and array-padding parameters for loops in numerical kernels (17). Waterman showed that binary search on the size could be effective (27). Another project that automatically tunes the optimization parameters to produce
high-quality code is the Automatically Tuned Linear Algebra Software (ATLAS) (29), which installs a library of basic linear-algebra subprograms on a given machine. The installation process uses robust search heuristics and extensive testing. It does this search on each installation, so it repeats a huge amount of work. The ATLAS kernel produces comparable quality code to those vendor-supplied, hand tuned codes. However, it only works on a small set of linear algebra libraries who behavior is well understood.

The algorithm used to tune parameters is unlikely to be effective on the transformation ordering problem. Parameter tuning is usually a combinatorial problem since the number of variables in the search space is fixed and there is no particular ordering between the parameters. However, the transformation ordering problem is very sensitive to the order of the transformations, and the length of the sequence is not restricted by the problem itself.
Chapter 3

Background

This chapter first presents the experimental framework, followed by the analysis of the optimization space and a review of current search algorithms.

3.1 Experimental Framework

3.1.1 Platform

Our compiler performs optimizations on the Intermediate Language for Optimizing Compilers (ILOC). ILOC looks like the assembly language for a low-level RISC machine and has a collection of features that simplify the construction of transformations and code generators. The performance of the code produced by our ILOC compiler using default settings on a SPARC machine is comparable to gcc with the -O2 flag.

The front end of our compiler accepts both C and Fortran programs and transforms them to ILOC intermediate representation. The back end of our compiler can generate the SPARC assembly code for the ILOC program. We also have an ILOC virtual machine which can run ILOC programs directly. In both cases, the program is
instrumented so that the actual number of instructions executed is used as the feedback metric for the compilation sequence. We use this metric rather than measuring the actual running time because the instruction count is precise and does not vary between runs. Minor variations in timing accuracy can perturb the search.

### 3.1.2 Transformation Set

The 16 transformations used in the thesis are listed in Table 3.1. They are low-level ILOC-based transformations which are widely implemented in both commercial and research compilers. Our implementation ensures that for each transformation, it can take any valid ILOC program as input and produce a valid ILOC program. This allows us to run the compilation transformations in arbitrary order, which is an important feature for an adaptive compiler.

Our transformations interact with each other. Some transformations overlap in the optimization that they perform. For example, DVNT value numbering (x) will kill the opportunities of EBB (y)/local value numbering (u). Some transformations, for example loop peeling or algebraic reassociation, may create opportunities for other transformations. Due to the interaction between transformations and differences in input programs, no single compilation sequence is good for all programs (16).
| c | Sparse conditional constant propagation (28) |
| d | Dead code elimination (12; 11) |
| g | Optimistic value numbering (3) |
| l | Partial redundancy elimination (21) |
| m | Renaming builds the name space needed by the implementations of l and z. The compiler inserts it automatically before l or z |
| n | Useless control-flow elimination— (11) |
| o | Peephole optimization of logically adjacent operations (14) |
| p | Peel the first iteration of each innermost loop |
| r | Algebraic reassociation (4) |
| s | Register-to-register copy coalescing (7) |
| t | Operator strength reduction (10) |
| u | Local value numbering (11) |
| v | Optimistic global value numbering (25) |
| x | Dominator tree value numbering (DVNT) (11) |
| y | Extended basic-block (EBB) value numbering (11) |
| z | Lazy code motion (18) |

**Table 3.1**: The Transformation Set of Our Adaptive Compiler

### 3.2 Optimization Space

Table 3.2 shows some characteristics of our optimization space. Characteristics of the optimization space depends on many settings including the transformation set, the benchmark chosen, the backend, and the objective function.

Columns 2 and 3 of Table 3.2 show the best sequences that we have found and their speedup over built-in sequence \texttt{rvzcdtvzcod}. Those best sequences are the result of running various search algorithms using multiple seeds and human tuning. They are not necessarily true global minima, but they are good criteria to evaluate the quality of sequences. Column 4 shows the speedup of the best sequences found by 1000 random probes.

Loop peeling is the most frequent transformation in the best sequences. It moves
<table>
<thead>
<tr>
<th>Program</th>
<th>Best Seq. Known</th>
<th>Best Seq. Speedup</th>
<th>Random Seq. Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmin</td>
<td>ppxlcnpnpod</td>
<td>1.15</td>
<td>1.13</td>
</tr>
<tr>
<td>zeroin</td>
<td>orvnpxypcod</td>
<td>1.10</td>
<td>1.07</td>
</tr>
<tr>
<td>svd</td>
<td>odgznvtdctpvn</td>
<td>1.15</td>
<td>1.10</td>
</tr>
<tr>
<td>spline</td>
<td>lrxotpppcdov</td>
<td>1.12</td>
<td>1.04</td>
</tr>
<tr>
<td>si</td>
<td>ztrcpppnpppppppgpodzp</td>
<td>1.49</td>
<td>1.27</td>
</tr>
<tr>
<td>fehl</td>
<td>rtmpppxrvpoc</td>
<td>1.13</td>
<td>1.06</td>
</tr>
<tr>
<td>bitcns</td>
<td>ppppppppppppppppppppppppppzdpp</td>
<td>1.68</td>
<td>1.26</td>
</tr>
<tr>
<td>sha</td>
<td>lpppppppppppppppppppppppppppc</td>
<td>2.06</td>
<td>1.43</td>
</tr>
</tbody>
</table>

Table 3.2: Characteristic of the Optimization Space.

the first iteration of a loop outside of the loop. Peeling a loop may create opportunities for other code-improving transformations (22). Loop unrolling is currently not in our pool, because it is difficult to identify a loop in a low level assembly-like language that does not carry loop notations.

Many groups are working on the transformation selection and ordering problem, but the characteristics of the optimization spaces appear to be very different (19; 20; 2; 1). As evidence of this, random probing has a very poor performance in our space, but it is reported in Agakov’s study (1) to achieve near maximum performance after 500 hundred trials. This implies good sequences are more sparse in our space than theirs. In VISTA (20), Kulkarni et al. exhaustively generated all function instances for most functions. However, this is nearly impossible in our system, because some transformations change the code aggressively. For example, applying loop peeling iteratively will generate a new version of the code until all iterations of the longest loop have been peeled.

17
Due to the difference between optimization spaces, comparing the absolute number, e.g. the maximum speedup, or number of trials conducted between systems is not an apples-to-apples comparison. It is possible that a search algorithm that performs well in one space does not perform well in other spaces. It remains unclear how the properties of an optimization space can affect the performance of a search algorithm.

3.3 Review of Current Search Algorithms

Four algorithms are implemented: random probing, random-restart hill climbing with patience (HC), genetic algorithms (GAs), and greedy constructor (GC). This section briefly reviews these four algorithms. The details of the search algorithm implementations can be found in Grosul’s Ph.D. thesis (16).

3.3.1 Genetic Algorithms

Genetic Algorithms (GAs) are global search heuristics that are inspired by evolutionary biology using concepts such as inheritance, mutation, selection, and crossover. GA is often used in search spaces where the properties of the space are not clear or the space is too complicated to use a more direct search method.

GA maintains a population of genetic representations, i.e., compilation sequences during the search progress. The initial generation of population is generated randomly. During each successive generation, a portion of the best population survives intact. The rest of the population is generated by mutation and crossover. The
parameter settings of the GAs are described in Chapter 6.

3.3.2 Hill Climbing Algorithms

Hill climbing algorithms attempt to find a local minimum by repeatedly descending from one point in the space to one of its neighbors that has a better fitness value. Hill climbing is a local search algorithm, because it makes its decisions on local information, i.e. its neighbors. In our space, neighbors are defined by Hamming distance: two neighbor sequences are of the same length and differ in exactly one position. Hill climbing algorithm will become stuck at a local minimum if all the neighbors of the current sequence have worse fitness values. The starting point of the hill climbing algorithm is very important to the quality of the local minimum where it terminates at last.

To compensate for the disadvantage of local decisions and the effect of bad starting points, the hill climbing algorithm is often restarted multiple times with random starting points. In practice, impatient random descent is used to reduce the time in exploring neighbors and thus allow more restarts given a certain time of budget. At a patience level of 10%, at most 10% of the total neighbors at evaluated before deciding the current point is a local minimum and the algorithm restarts.

3.3.3 Greedy Construction

The greedy construction algorithm (GC) extends a good sequence by one character at a time. Each time, all combinations of transformations and positions for insertion
are considered, and the best one is selected. The algorithm terminates when there is no further improvement can be found.

3.3.4 Discussion

Both genetic and hill climbing algorithms are reported to find good sequences in small and large spaces if we are willing to perform thousands of trials. However, in the even larger space with no sequence length limitations, the performance of both GAs and HCs are not satisfactory. According to the experimental results in Figures 6.1 to 6.4, GAs on average achieve less than 70% of the maximum possible improvement after 1000 trials. The hill climbing algorithm seems to be better on average, but for some length settings, it is still below 50% on certain benchmarks.

We believe the reason for the poor performance of GAs is the following. The mutation and 1-point crossover operations over fixed-length sequences of compiler optimizations do not explore the space of compiler optimization sequences effectively. The mutation operator, which randomly generates point variations of a sequence, makes local changes very slowly, since mutation rates are generally set low. The 1-point crossover operator is quite destructive and does not generate semantically meaningful variations. The constraint of fixed-length sequences limits the range of performance gains that can be achieved. Some benchmarks need fairly short compilation sequences, and so a fixed-length GA spends valuable time and resources learning no-op subsequences to pad the sequence to the full pre-determined length. Other benchmarks need longer sequences, and the GA fails to realize the full benefit of the
range of optimizations available in the compiler. The deficiencies of the operators used to generate variations make genetic algorithms slow in finding high-quality sequences. Attempts to fix this inherent weakness in exploration strategy by tuning algorithm parameters such as the population size, or the size of the fixed-length representation, only yield limited improvement in search performance as shown in our experiments.

The better performance of hill climbers over genetic algorithms suggests the possibility that local searches might work well for this problem. Besides, according to the experiments, the length settings for a hill climber can make a huge difference, with a margin of over 40% on some benchmarks. However, both Grosul's GA and his HC algorithms work on fixed-length framework. The length of the sequences they work on is predetermined before the search starts and is not flexible while searching. This motivates the work to design a new local-search algorithm that can search for sequences of variable length. The GNE algorithm proposed in this thesis has both features.

As pointed out in Grosul's thesis, the critical shortcoming of the greedy construction algorithm is its lack of ability to deal with enabling/enabled transformations pairs. More details are discussed in Section 4.3.
Chapter 4

GNE: Greedy Neighbor Exploration

This chapter describes the new local-search algorithm, GNE. It first presents the outline of the search algorithm and then describes the key procedure: neighbor exploration. The last section discusses the improvement of GNE over the original greedy construction algorithm.

4.1 Outline

The outline of GNE is shown in Figure 4.1. The algorithm runs like rounds of tournament. Each round consists of five stages: generation, qualification, training, competition, and award.

**Generation:** Random sequences are generated.

**Qualification:** The qualification round pre-screens the random sequences generated.

Each generated sequence will first go through a quick *cleanup* procedure, which removes redundant and negative transformations. The best sequence after cleanup will advance to the training phase, and the remaining sequences are discarded. However, it is worth the extra cost to pre-screen the sequences so
that we have a better starting point than random sequences for the neighbor exploration in the training phase.

Training: In the training phase, the neighbor space of the best sequence generated this round is explored. The best sequence found during the exploration is declared the challenger of the round and will be compared against the current champion. The details of the neighbor exploration is described in Section 4.2.

Competition: The challenger of the round is compared with the current champion. Whoever has a better fitness value wins the competition.
**Award:** If the current champion wins this round, it is awarded one iteration of neighbor exploration. Each new iteration of the neighbor exploration is likely to find better sequences because each iteration is likely to find new space that was not explored in previous iterations. To better use our limited budget, instead of exhaustively exploring the neighbor space of any sequence, we split the effort into iterations and award one new iteration to the winner of the round. If the champion loses, it is replaced by the challenger of this round and a better sequence is found.

GNE repeats the tournament round after round indefinitely, though in practice most improvement appears in the first 10 rounds. In our experiment, the search process terminates after 1000 trials, which allows 9-10 rounds for most benchmarks.

The rest of this section will describe the key part of GNE: neighbor exploration.

### 4.2 Neighbor Exploration

The definition of neighbor is fundamental to search algorithms. In the fixed-length framework used by both GA and HC, it is convenient to define neighbors by Hamming distance: two sequences of the same length are considered neighbors if they differ in exactly one character. The experiments in Section 6.1 show that the hill climbers using Hamming distance as the definition of neighbors do not deliver satisfactory results.

This thesis defines the neighbor by editing distance, i.e., two sequences are con-
sidered neighbors if one can be derived from another by inserting or removing one transformation. Thus, two neighbors in Grosul’s metric have an editing distance of two and are not considered neighbors in our metric. Two procedures are used to explore the neighbor space from a starting point: cleanup and optimistic greedy construction.

4.2.1 Cleanup

The cleanup procedure removes transformations that are redundant or negative to the quality of the given sequence. A transformation in the sequence is redundant if deleting the transformation from the sequence does not affect the quality of the sequence, or negative if deleting the transformation from the sequence produces a better sequence.

Both greedy construction and random sequence generation can create redundant or negative transformations. Even if a transformation is not redundant or negative at the time, it may become redundant or negative if some other transformations are inserted into the sequence. For example, inserting a more powerful value numbering pass may make a less powerful value numbering pass redundant; if two transformations interact with each other by one transformation enabling the other, missing either one of the two transformations may make the other transformation redundant or negative.

There are two main purposes to cleaning up a sequence: First, better sequences may be found by removing negative-effect transformations. Second, by removing redundant transformations, sequences are kept short, so that the greedy construction
process can be iterated to find better sequences. Without the cleanup process, the length of the sequence will become increasingly long and the compilation cost of a single trial will increase significantly.

Algorithm 1 shows the pseudo-code for the cleanup procedure. The main body of the cleanup procedure is a loop that iterates round by round. At each round, a transformation is randomly picked to be dropped. If the residual sequence has an equal or better fitness value, the transformation is removed from the sequence. Otherwise, the transformation is marked and will not be considered for dropping in the future. The cleanup procedure terminates when all transformations in the sequence are marked. Cleaning up a sequence of length \( l \) will take at most \( l \) trials. Our cleanup algorithm does not necessarily find the best or the shortest subsequence, but it serves our purposes efficiently.

**Algorithm 1 Cleanup**

*Input*: sequence  
unmark all transformations in *sequence*  
repeat  
Randomly pick an unmarked transformation at position \( p, t_p \), from the sequence  
\[ \text{new}_s \leftarrow \text{sequence} \text{ with } t_p \text{ dropped} \]  
evaluate \[ \text{new}_s \]  
if *sequence* is equal or better than *new*_s then  
set \( t_p \) as marked  
else  
//\( t_p \) dropped  
\[ \text{sequence} \leftarrow \text{new}_s \]  
end if  
until All transformations in *sequence* are marked  
*return* *sequence*
4.2.2 Optimistic Greedy Construction

Algorithm 2 shows the pseudo-code for the greedy construction algorithm. The algorithm extends the base sequence one transformation at a time. At each step, one transformation is picked randomly and inserted into the position that delivers the most improvement, with ties broken randomly. If a transformation cannot deliver improvement, it is still inserted into the best position but is marked, so that it will not be selected for insertion in the future. When all transformations are marked or the length of the sequence exceeds the maximum length limit, the algorithm stops inserting additional transformations. The greedy construction procedure finishes after unmarking all transformations, and calls the cleanup procedure to remove redundant transformations in the sequence.

Figure 4.2 illustrates the neighbor exploration process (or training phase) in GNE. The sequence under the transition bar(=>) is the current sequence of which the neighbor set is to be explored. The neighbor set of the current sequence is listed between the transition symbol. The sequence with a plus indicates an improvement over the current sequence. The sequence with an equal indicates the sequence has the same fitness value as the current sequence. At each step, the best neighbor becomes the current sequence. Note that although there is no improvement in step 2 when inserting the peephole optimization(o), our algorithm continues, and later, a better sequence vyod is found. Since global value numbering(y) takes care of all the opportunities of the EBB value numbering y, y is now redundant and removed by the
Algorithm 2 Optimistic Greedy Construction

Input: sequence

// seq is the sequence we work on, sequence is the best that we encounter and return
seq ← sequence
repeat
    Pick an unmarked transformation t
    new_s ← best sequence after insertion t into random selected positions of seq, breaking
ties randomly
    if new_s is better than sequence then
        sequence ← new_s
    end if
    if new_s is not better than seq then
        set t as marked
    end if
    seq ← new_s
until All transformations are marked or seq is too long
cleanup(seq)
unmark all transformations
if seq is better than sequence then
    sequence ← seq
else
    cleanup(sequence)
end if
return sequence

cleanup procedure.

For a pool of $t$ transformations, and a sequence of length $l$, there are at most
$t \times (l + 1)$ neighbor sequences for greedy construction. However, in practice, only a
small number of positions are selected for insertion given a transformation $x$. This
usually gives us satisfactory results, because inserting $x$ before or after transformation
$y$ will produce the same code, unless $x$ and $y$ interact. Thus there is a high probability
to produce a sequence with the best fitness value for insertion even if a small portion
of positions are selected.
4.3 Discussion

The idea of greedy construction was first proposed in Grosul’s work (16). However, the algorithm is reported to be inferior to other search algorithms such as genetic algorithms and hill climbing due to the uncharacterized interaction between transformations, especially when a transformation requires an enabling transformation to be beneficial. The greedy construction algorithm in GNE improves Grosul’s algorithm and is more effective at finding better sequences by extending the base sequence. The improvement comes from the following aspects:

1. Grosul’s greedy construction algorithm only inserts transformations, but will not remove them. If a mistake is made by inserting a bad transformation, we will never have a chance to fix the mistake. However, our greedy construction interacts with the cleanup procedure, so redundant or negative transformations inserted by the greedy construction procedure will be removed by cleanup.

29
2. The amount of improvement one can achieve in a search algorithm such as hill climber depends heavily on the starting point. In Grosol's GC algorithm (16), the base sequence is the empty string. However, in this paper, the base sequence has already survived the qualification phase. Thus, we are starting from a relatively good sequence.

3. In this thesis, the greedy construction algorithm takes an optimistic approach. If a transformation does not deliver improvement, it is not discarded immediately. Instead, we mark the transformation as not profitable and leave it in the sequence, either hoping it will interact with some transformation in the future to deliver improvement or be removed during the cleanup procedure. Transformations marked not profitable will not be picked in the future until all transformations are marked as not profitable. Then we run simplification on the sequence to remove those redundant or negative transformations and all marks are reset.

The optimistic approach has an advantage over simple greedy construction when two transformations $B$ or $C$ do not improve the sequence individually but will deliver improvement when used together. Such a situation is common in the implementation of transformations: for example, our implementation of peephole optimization($o$) will not deliver improvement to code unless there is a followup dead code elimination($d$) because peephole optimization leaves dead code in the program, assuming a followup run of dead code elimination will remove them. For the example shown in Figure 4.2,
Grosul’s greedy construction algorithm will quit after getting \( vy \), because neither \( o \) nor \( d \) can provide improvement over the sequence \( vy \) if inserted individually. The optimistic approach inserts \( o \) into the sequence even if there is no improvement and assumes the later cleanup procedure will remove it if it is not finally profitable. Finally, \( o \) interacts with \( od \) and a better sequence \( vyod \) is found.
Chapter 5

Code Normalization and Redundancy Removal

It has been reported in Kulkarni et al.'s study (19) and confirmed in our system that programs compiled by different sequences can turn out to be equivalent, which means two programs have only trivial differences and will have the same runtime instruction count. This can happen when two transformations that do not interact are applied in a different order, or a combination of different transformations have the same effect on the program. This chapter describes the techniques to normalize programs and detect equivalent code.

5.1 Code Normalization

To detect equivalent programs, the first step is to normalize the program. Normalization is a transformation such that if two programs are identical after being normalized, they are equivalent. The simple code normalization scheme is very easy to implement. It simply removes all comments and NOPs, maps labels and registers to new names. Table 5.1 shows the sample of simple code normalization. The left column (Code A) is the code before normalization and the right column is the code after
normalization. Labels are renamed; comments are removed; registers are mapped to new names: r2 is mapped to r1, r2 is mapped to r2, etc.

<table>
<thead>
<tr>
<th>Code A</th>
<th>Normalized (Simple)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LL01:</td>
<td>L1:</td>
</tr>
<tr>
<td>r2+r4=&gt;r5</td>
<td>r1+r2=&gt;r3</td>
</tr>
<tr>
<td>r4*r5=&gt;r1</td>
<td>r2*r3=&gt;r4</td>
</tr>
<tr>
<td>r5=&gt;r10</td>
<td>r3=&gt;r5</td>
</tr>
<tr>
<td>#some comments</td>
<td>r4=&gt;r6</td>
</tr>
<tr>
<td>r1=&gt;r8</td>
<td>jmp L3</td>
</tr>
<tr>
<td>jmp L3</td>
<td>L2:</td>
</tr>
<tr>
<td>LL02:</td>
<td>r1-r2=&gt;r7</td>
</tr>
<tr>
<td>r2-r4=&gt;r11</td>
<td>r1*r2=&gt;r8</td>
</tr>
<tr>
<td>r2*r4=&gt;r7</td>
<td>r7=&gt;r5</td>
</tr>
<tr>
<td>r11=&gt;r10</td>
<td>r8=&gt;r6</td>
</tr>
<tr>
<td>r7=&gt;r8</td>
<td>jmp L3:</td>
</tr>
<tr>
<td>jmp L3:</td>
<td>L3:</td>
</tr>
<tr>
<td>LL03:</td>
<td>r5+r6=&gt;r9</td>
</tr>
<tr>
<td>r10+r8=&gt;r9</td>
<td>...</td>
</tr>
</tbody>
</table>

**Figure 5.1: Simple Code Normalization - Code A**

The effectiveness of the simple normalization differs significantly from system to system. VISTA uses the simple code normalization scheme to detect equivalent code, and reports 87% of the programs evaluated by GA are equivalent (19). However, using their normalization technique in our system finds less than 5% of the programs are equivalent.

A large amount of the difference is due to the register-to-register copy instructions at the end of the basic blocks. For example, code A in Figure 5.1 and code B in Figure 5.2 are almost identical except for the order of the two copy instructions at the end of the basic block L1. However, simple code normalization will treat them
as distinct programs. Such a situation is very common in our compiler. Many of
our transformations are SSA based (13), for example, various value numbering passes
(v,x,y,g), constant propagation (c), etc. After the optimization is finished, the SSA
form is deconstructed (5). The deconstruction process deletes the φ-nodes at the
beginning of the basic block and appends the corresponding register-to-register copy
instructions to the end of precedent basic blocks. Since in SSA form, the φ-nodes are
considered to be executed simultaneously and copy instructions may be appended in
an arbitrary order. This causes difficulties for equivalent-code detection.

\[
\begin{array}{ll}
\text{LL01:} & \ldots \\
\text{r10+r4=>r5} & \text{L1:} \\
\text{r4*r5=>r6} & \text{r1+r2=>r3} \\
\text{r6=>r1} & \text{r2*r3=>r4} \\
\text{r5=>r6} & \text{r4=>r5} \\
\text{jmp L3} & \text{r3=>r6} \\
\text{LL02:} & \text{jmp L3} \\
\text{r10-r4=>r8} & \text{L2:} \\
\text{r10*r4=>r7} & \text{r1-r2=>r7} \\
\text{r7=>r6} & \text{r1*r2=>r8} \\
\text{r8=>r1} & \text{r8=>r5} \\
\text{jmp L3: #some comments} & \text{r7=>r6} \\
\text{LL03:} & \text{jmp L3:} \\
\text{r1+r6=>r9} & \text{L3:} \\
\ldots & \text{r5+r6=>r9} \\
\ldots & \ldots \\
\text{Code B} & \text{Normalized (Simple)}
\end{array}
\]

**Figure 5.2: Simple Code Normalization - Code B**

To detect this situation, the simple normalization is extended to work on the
SSA form of the program. When constructing the SSA form, copy instructions are
propagated and removed. Then the normalization process sorts the φ-nodes at the
beginning of each basic block. To determine the order of φ-nodes, each φ-node is
<table>
<thead>
<tr>
<th>SSA form for Code A</th>
<th>SSA form for Code B</th>
<th>Normalized (SSA-based)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1: #5</td>
<td>L1: #5</td>
<td>L1: #5</td>
</tr>
<tr>
<td>r1+r2=&gt;r3</td>
<td>r1+r2=&gt;r3</td>
<td>r1+r2=&gt;r3</td>
</tr>
<tr>
<td>r2*r3=&gt;r4</td>
<td>r2*r3=&gt;r4</td>
<td>r2*r3=&gt;r4</td>
</tr>
<tr>
<td>jmp L3</td>
<td>jmp L3</td>
<td>jmp L3</td>
</tr>
<tr>
<td>L2: #5</td>
<td>L2: #5</td>
<td>L2: #5</td>
</tr>
<tr>
<td>r1-r2=&gt;r5</td>
<td>r1-r2=&gt;r5</td>
<td>r1-r2=&gt;r5</td>
</tr>
<tr>
<td>r1*r2=&gt;r6</td>
<td>r1*r2=&gt;r6</td>
<td>r1*r2=&gt;r6</td>
</tr>
<tr>
<td>jmp L3:</td>
<td>jmp L3:</td>
<td>jmp L3:</td>
</tr>
<tr>
<td>L3: #...</td>
<td>L3: #...</td>
<td>L3: #...</td>
</tr>
<tr>
<td>phi r3 f5 =&gt; r7</td>
<td>phi r4 r6 =&gt; r7</td>
<td>phi r3 f5 =&gt; r7</td>
</tr>
<tr>
<td>phi r4 r6 =&gt; r8</td>
<td>phi r3 r5 =&gt; r8</td>
<td>phi r4 r6 =&gt; r8</td>
</tr>
<tr>
<td>r7+r8=&gt;r9</td>
<td>r8+r7=&gt;r9</td>
<td>r7+r8=&gt;r9</td>
</tr>
</tbody>
</table>

**Figure 5.3**: Code A and Code B's SSA form and their Normalized SSA form

encoded to a string. The order of φ-nodes should be solely determined by the actual instructions and the def-use chain, and should not depend on the order of any φ-nodes in the same basic block. The pseudo-code in Figure 5.4 is used to encode φ-node P.

Figure 5.4 shows an example of the encoding of the φ-node. The cycles indicate the φ-nodes and the squares indicate registers defined by instructions other than φ-nodes. The numbers in the cycle are the timestamps.

Figure 5.3 shows the SSA form of code A and code B and the code after normalization. The SSA form of code A and code B differ in the order of the φ-nodes. The normalization will sort the φ-nodes according to the encoding, and yields identical code for code A and B.
String Encode(PhiNode P)
{
    timestamp(P) <= count++; // count is a global counter initialized to 1
    s <= '\n';
    for (i=1; i<=P.number_of_parameters; i++) {
        p <= P's ith parameter;
        if (p is not a phi-node) {
            s = s + String(p);
        } else {
            if (timestamp(p)==0)
                s = s + Encode(p);
            else // backedge
                s = s + timestamp(p);
        }
        if (i < m)
            s = s + ',
    }
    return s+'\n';
}

![Diagram of a tree with nodes 1, 2, 3, and 1, r1, r2, r3, and a label (1,r1,r2),(2,r2,r3)](image)

**Figure 5.4: φ-Node Encoding**

The static instructions counts for each basic block in the original code are included in the normalized program as comments after basic block’s label. If two normalized programs match, the instruction count of each basic block before the copy instructions are coalesced during SSA construction should also match assuming the same input data.
5.2 Redundancy Removal

Redundancy is detected and avoided by hashing. Our system has two hash tables: one for compilation sequences and one for programs. For programs, it is impractical to hash the whole text due to both the huge space requirement and the time it takes to go over the hash table to find a match. VISTA uses a cyclic-redundancy check (CRC) algorithm (6) to compute the hash key of the problem. However, when I tried CRC-32 checksum in our system, the false collision rate is high and sometimes the system mistakenly treats the code produced by a better sequence as the same as some version previously produced. The reason is not clear, but my speculation is that CRC was originally designed to detect errors that occurred during data transmission and were caused by noise; however, the behavior of different transformations on the code is very different from noise.

This work uses the MD5 algorithm (23) as the hash function to compute the hash key. A false collision of MD5 is much harder to generate since the hash key is 128-bits, and only a few thousands of programs are generated.

As shown in Figure 5.5, when a sequence is submitted for evaluation, it is first looked up in the sequence hash table to see if it has been seen before. If it is a miss, the program is compiled using the sequence, and dormant transformations are identified. A transformation is dormant if the programs before and after its application are equivalent. After removing the dormant transformations from the sequence, the sequence is searched in the sequence hash table again. If still not found, the output
Figure 5.5: Hashing and Redundancy Removal

program is normalized, and the hash key of the program is computed. If the key hits the program hash table, it assumes that an equivalent program has already been evaluated, and the measurement of last time is returned. If all hash table checks miss, the program is run and the measurement is returned.

5.3 Experiment

Table 5.1 shows the percentage of the number of runs saved by the equivalent-code detection techniques using different search algorithms. For our GAs, on average, 27% of requests produce equivalent programs, which is still much less than the number reported in VISTA even with the enhanced code normalization. Since our equivalent-code detection technique can find all the equivalent programs found by VISTA, this implies that our transformation set and our search algorithms can produce a greater
<table>
<thead>
<tr>
<th>Program</th>
<th>GA-15</th>
<th>GNE</th>
<th>Random</th>
<th>Program</th>
<th>GA-15</th>
<th>GNE</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmin</td>
<td>18%</td>
<td>47%</td>
<td>9%</td>
<td>zeroin</td>
<td>56%</td>
<td>61%</td>
<td>13%</td>
</tr>
<tr>
<td>svd</td>
<td>12%</td>
<td>35%</td>
<td>0%</td>
<td>spline</td>
<td>24%</td>
<td>40%</td>
<td>1%</td>
</tr>
<tr>
<td>si</td>
<td>35%</td>
<td>55%</td>
<td>5%</td>
<td>fehl</td>
<td>17%</td>
<td>47%</td>
<td>3%</td>
</tr>
<tr>
<td>sha</td>
<td>36%</td>
<td>39%</td>
<td>6%</td>
<td>bitcnts</td>
<td>21%</td>
<td>41%</td>
<td>5%</td>
</tr>
</tbody>
</table>

Table 5.1: Percentage of Equivalent Code Detected

number of distinct versions of intermediate code on the benchmarks. As shown in Table 5.1, GNE generates more equivalent programs than GAs. This is just as expected, since GNE puts most of the effort into exploring the space of neighbors, in which many programs are equivalent.
Chapter 6

Experimental Results

This chapter presents the experimental results for genetic algorithms, hill climbing algorithms and GNE when used to search for a good compilation sequence. The genetic and hill climbing algorithms are described in Chapter 3. GNE is described in Chapter 4. The first part of this chapter examines the performance of GAs and HCs using varying length parameters. The second part of the chapter compares GA and HC with GNE.

The objective of the experiment is to optimize code speed which is measured by the count of instructions executed at runtime. The backend of the compiler used in the experiment is SPARC.

For the graphs in this chapter, the performance of a search algorithm is illustrated by the number of trials conducted along the x-axis and the corresponding percentage of the maximum improvement achieved on the y-axis. The number of trials is the count of actual runs of the program, not including those requests that are avoided by equivalent code detection described in Chapter 5. We show the results for the first 1000 trials, which we believe is the high-end of the number of trials affordable for
most programs. The percentage of the maximum improvement achieved is computed based on the fitness value of the built-in sequence *ruvcodtuzcod* and the best sequence found.

<table>
<thead>
<tr>
<th>Program</th>
<th>Suite</th>
<th>Insts</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmin.f</td>
<td>fmm</td>
<td>267</td>
<td>finds the minimum value of a function over an interval</td>
</tr>
<tr>
<td>zeroin.f</td>
<td>fmm</td>
<td>217</td>
<td>find a zero of a user function in an interval</td>
</tr>
<tr>
<td>svd.f</td>
<td>fmm</td>
<td>2122</td>
<td>singular value decomposition</td>
</tr>
<tr>
<td>spline.f</td>
<td>fmm</td>
<td>1037</td>
<td>compute the coefficients for a cubic interpolating spline</td>
</tr>
<tr>
<td>si.f</td>
<td>spec</td>
<td>131</td>
<td>hottest function of duduc</td>
</tr>
<tr>
<td>fehl.f</td>
<td>spec</td>
<td>567</td>
<td>fourth-fifth order runge-kutta method</td>
</tr>
<tr>
<td>bitcns.c</td>
<td>mibench</td>
<td>159</td>
<td>test program for bit counting functions</td>
</tr>
<tr>
<td>sha.c</td>
<td>mibench</td>
<td>564</td>
<td>secure hash algorithm</td>
</tr>
</tbody>
</table>

**Table 6.1: Benchmarks used in the Experiments**

The benchmarks used in the experiments are listed in Table 6.1. The benchmarks are selected from three standard benchmark suites: FMM, SPEC95, and MiBench. Since each benchmark will be compiled and run thousands of time in our experiments, large programs that take a long time to compile and run are not included in our test suites.

### 6.1 Genetic Algorithms and Hill Climbing

For the GAs, Grosul's optimized parameter settings are used. Different lengths for the genetic representations are tried. The results are shown as GA-15, GA-20 and GA-25. The parameters of GA is described below:
Length: The length of the sequences: 15, 20 or 25.

Population: Each generation consists of 50 strings.

Elitism: The best 10% always survive intact.

Reproduction: The remaining 90% are created with the crossover mechanism.

Selection: Strings are selected for crossover at random with fitness-biased weights.

Crossover: It uses a single-point random crossover.

Mutation: Strings created by crossover are subject to position-by-position mutation, with probability 0.02.

Duplication: Duplicates are mutated into untested strings.

The patience of the hill climbing algorithm is set to 10%; i.e., at each step, only 10% of the neighbors are randomly picked and evaluated. The length of the representations varies from 15 to 25, and the results are shown as HC-15, HC-20 and HC-25.

The search performance of the genetic algorithms and hill climbing algorithms on the SPARC backend are shown in Figure 6.1 to Figure 6.3. The average over all benchmarks is shown in Figure 6.4.

The variation of sequence length for the genetic algorithm has little effect on average performance, with GA-20 and GA-25 generally better than GA-15. For an individual benchmark, the performance gap between the best and the worst GA can be as large as 15%. It is hard to decide which length is the best for all. For example, GA-25 outperforms GA-20 and GA-15 by at least 30% in fehl, but is excelled by GA-20 by 10%-20% in zeroin and spline.
Figure 6.1: Genetic and Hill Climbing Algorithm for Benchmark fmin, zeroin, svd
Figure 6.2: Genetic and Hill Climbing Algorithm for Benchmark spline, si, fehl
Figure 6.3: Genetic and Hill Climbing Algorithm for Benchmark bitcnts, sha

For hill climbing, HC-25 appears to be the best length setting for most programs; however the improvement achieved is still less than 85% on average after 1000 trials.

For a specific benchmark, if the length parameter of GA or HC is set too long, they waste the time to fill the sequence with those no-effect transformations. Besides, the size of the neighbor set of a given sequence also grows exponentially to the length of the sequence. More neighbors need to be evaluated at each step if the density of better neighbors remains the same. For HC, this implies more trials are required to
hit the local minimum. The reason that HC-25 works well even for those programs whose best sequence is much shorter, for instance svd, is because of the distribution of the quality of the local minima. If the sequences of exceptional quality is very scarce, though HC with short length settings restarts more times and hits more local minima than HC-25, it cannot hit a better local minimum. For GA, because the population size is fixed, it may fail to find a better neighbor. Thus GA is more sensitive to its length setting. On the other hand, if the length settings is too short, we miss the opportunity to find the best sequence.

6.2 Comparing Searching Performance

In this section, the genetic and hill climbing algorithms are compared to GNE.

There are a few parameters to be decided for GNE. The parameters that give the best results on average are used in the comparison: each round, five sequences
are generated and only one sequence will pass the qualification; When inserting a transformation during the greedy construction, four random positions are chosen for insertion, and the transformation is inserted into the one that delivers the best fitness value with ties broken randomly.

Figures 6.5 and 6.6 compare the searching performance between GAs, HCs, and GNE, and the performance averaged on all benchmarks are shown in Figure 6.7. Among all three length settings of GAs and HCs, the best setting is used for each benchmark, although the best setting is not known ahead of time in practice.

The results show that GNE finds better sequences than HC and GA after about 200-300 trials. After 1000 trials, GNE outperforms the optimized hill climbing algorithm by 10% of the maximum available improvement, and outperforms the genetic algorithm by 20%. The number is averaged over all eight benchmarks. GNE not only outperforms the other two algorithms on average, it wins for almost all eight benchmarks except svd. For svd, GNE loses to HC-20 by only less than 1%.

Snapshots of the search process for each benchmark after 100, 200, 400 and 800 evaluations are shown in Table 6.2. The value shown for GA is the best of GA-15, GA-20 and GA-25 and the value shown for HC is the best of HC-15, HC-20 and HC-25. The actual speedup over the built-in sequence is shown. The average speedup of GNE after 100 and 200 trials is 1.18 and 1.22, and HC has comparable speedup at 1.17 and 1.22 respectively. GNE begins to beat HC after 200 trials. At 400 and 800 trials, the speedup of GNE is 1.31 and 1.34, while the speedup of HC is 1.25 and
Figure 6.5: Search Algorithms Comparison for Benchmark
Figure 6.6: Search Algorithms Comparison for Benchmark - cont.

Figure 6.7: Average Search Algorithms Performance

1.28.

Table 6.3 shows the number of trials required to achieve a certain amount of improvement. On average, it takes GNE 341 trials to achieve 80% of the maximum improvement available, while the same amount of improvement takes HC 817 trials to achieve and GA fails to achieve that mark within 1000 trials. GNE achieves the 90% mark for all benchmarks except spline while HC failed to reach the 90% mark.
<table>
<thead>
<tr>
<th>SPARC</th>
<th>100 Trials</th>
<th>200 Trials</th>
<th>400 Trials</th>
<th>800 Trials</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark</td>
<td>GNE</td>
<td>GA</td>
<td>HC</td>
<td>GNE</td>
</tr>
<tr>
<td>fmin</td>
<td>1.12</td>
<td>1.11</td>
<td>1.14</td>
<td>1.15</td>
</tr>
<tr>
<td>zeroin</td>
<td>1.07</td>
<td>1.06</td>
<td>1.06</td>
<td>1.07</td>
</tr>
<tr>
<td>svd</td>
<td>1.11</td>
<td>1.05</td>
<td>1.11</td>
<td>1.12</td>
</tr>
<tr>
<td>spline</td>
<td>1.05</td>
<td>1.01</td>
<td>1.04</td>
<td>1.06</td>
</tr>
<tr>
<td>si</td>
<td>1.30</td>
<td>1.24</td>
<td>1.25</td>
<td>1.32</td>
</tr>
<tr>
<td>fehl</td>
<td>1.04</td>
<td>1.04</td>
<td>1.06</td>
<td>1.06</td>
</tr>
<tr>
<td>bitcnts</td>
<td>1.29</td>
<td>1.22</td>
<td>1.27</td>
<td>1.37</td>
</tr>
<tr>
<td>sha</td>
<td>1.44</td>
<td>1.38</td>
<td>1.43</td>
<td>1.59</td>
</tr>
<tr>
<td>avg</td>
<td><strong>1.18</strong></td>
<td><strong>1.14</strong></td>
<td><strong>1.17</strong></td>
<td><strong>1.22</strong></td>
</tr>
</tbody>
</table>

Table 6.2: Speedup over built-in sequence of the best sequence found by GNE, GA and HC after 100, 200, 400 and 800 trials.

<table>
<thead>
<tr>
<th>SPARC</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark</td>
<td>GNE</td>
<td>GA</td>
<td>HC</td>
</tr>
<tr>
<td>fmin</td>
<td>70</td>
<td>109</td>
<td>47</td>
</tr>
<tr>
<td>zeroin</td>
<td>181</td>
<td>424</td>
<td>491</td>
</tr>
<tr>
<td>svd</td>
<td>83</td>
<td>377</td>
<td>44</td>
</tr>
<tr>
<td>spline</td>
<td>542</td>
<td>1000+</td>
<td>370</td>
</tr>
<tr>
<td>si</td>
<td>256</td>
<td>979</td>
<td>865</td>
</tr>
<tr>
<td>fehl</td>
<td>323</td>
<td>756</td>
<td>363</td>
</tr>
<tr>
<td>bitcnts</td>
<td>270</td>
<td>1000+</td>
<td>478</td>
</tr>
<tr>
<td>sha</td>
<td>229</td>
<td>1000+</td>
<td>608</td>
</tr>
<tr>
<td>avg</td>
<td><strong>238</strong></td>
<td><strong>851</strong></td>
<td><strong>254</strong></td>
</tr>
</tbody>
</table>

Table 6.3: Number of trials required to achieve 70%, 80% and 90% of the potential improvement on SPARC

### 6.3 Conclusion

The following conclusions can be experimentally drawn:

1. Genetic Algorithms do not explore the optimization space effectively. Changing the length parameters has only limited improvement.
2. The Hill Climbing Algorithm on average performs better than Genetic Algorithms. Generally, HC works better on longer length settings. However, the overall performance of HC is still not satisfactory. For some benchmarks, for example sna, HC achieves less than 70% of the total improvement possible.

3. GNE outperforms both GAs and HCs consistently on virtually all benchmarks. GNE automatically adapts to the length of the best sequence. It removes transformations from the sequence if they are found redundant, and inserts transformations into the sequence if found beneficial.
Chapter 7

Conclusion and Future Work

This thesis proposed two approaches to reduce the cost of adaptive compilation. First, we showed a local search algorithm, GNE, which can effectively explore the optimization space by adding and removing transformations. Second, we developed an SSA-based code normalization technique that can help detect and reduce redundancies while evaluating sequences in an adaptive compiler.

Experimental results show that current search algorithms do not explore the optimization space effectively. Different length settings for GA and HC can have different search performance. Our GNE outperforms both GAs and HCs by automatically adapting to the length of the best sequence.

Experiments also show that with the help of our SSA-based code normalization technique, 29% of the evaluations can be saved during the searches using GAs, and 46% for searches using GNE.

Combining these two techniques, we can now achieve 90% of the total possible improvement within 200-300 trials, which is less than a quarter of the efforts required by previous methods. Further improvement is possible by pursuing the following
directions:

- The search algorithms studies in this thesis do not bias individual transformations, although different transformations have different effect on the code and it is desirable to treat them differently when searching for a good compilation sequence. For example, some transformations, for example loop peeling, are more likely to appear multiple times than other transformations. Some transformations, such as dead code elimination, are more likely to appear after other transformations, such as effective partial redundancy elimination, while is more likely to appear in the beginning of a good sequence.

  Offline learning of good sequences may produce a statistical model to develop transformation-biased search algorithms, which have the promise to find a good sequence faster.

- This thesis does experiments on the same 16 transformations pool. It is interesting to see how the performance of current search algorithms changes for a different set of transformation with different properties. Sandoval has done some experiments to compare the performance of our adaptive compiler after adding a loop unrolling transformation to the pool (24).
Bibliography


[23] Rfc 1321 - the md5 message-digest algorithm.


[26] Triantafyllis, S., Vachharajani, M., Vachharajani, N., and August,


