Compilation Order Matters*

Keith D. Cooper, Timothy J. Harvey, Devika Subramanian, and Linda Torczon
Department of Computer Science
Rice University
Houston, TX, USA

ABSTRACT

At design time, the compiler writer selects a set of optimizations and an order in which to apply them. These choices have a direct impact on the quality of code that the compiler can generate. They also determine, to a great extent, the set of programs for which the compiler generates good code. Making good choices is difficult for several reasons. The sheer number of transformations that have been described in the literature is daunting. The relationship between features of the input program and improvements (or degradations) produced by a transformation is not formally characterized. The interactions between these transformations are poorly understood. Finally, many of the existing transformations have overlapping effects, either singly or in combination.

We have built a prototype adaptive compiler that changes its selection of transformations and its application order based on the input program, the performance characteristics of the target machine, and an explicit external objective function. This adaptive compiler performs a series of experiments to find a sequence of optimizations that minimizes the objective function. This prototype compiler lets us explore the impact of transformation selection and ordering on the code that the compiler produces. This paper describes our prototype adaptive compiler. It presents experimental results that demonstrate the impact of program specific optimization sequences. It begins to characterize the search space in which the adaptive compiler operates.

1. INTRODUCTION

The Fortran Automatic Coding System, released in 1957, was one of the earliest compilers [2]. It established a model for organizing compilers as a series of linear passes, shown in Figure 1. The first pass, or front end, dealt with the source language and converted it into an internal representation (IR) for subsequent passes. The next two passes, called the optimizer or middle end, analyzed and transformed that IR form of the program. The final three passes transformed the optimized IR program into code for the target machine—in this case, the IBM 704.

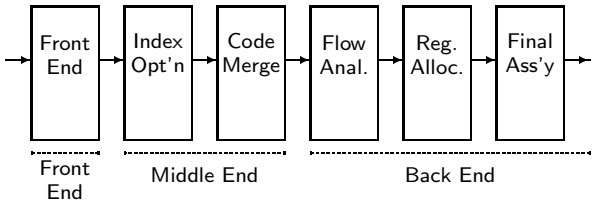


Figure 1: The Fortran Automatic Coding System

Open up a modern compiler and you will find this same basic structure. It may have more passes; for example, the documentation for the SUIF 1 system listed eighteen distinct transformations [15]. It may have several front ends; for example, both SUIF 2 and the SGI PRO64 compiler provide front ends for Fortran, C, C++, and Java. It may have several different IRs, or a single IR whose level of abstraction is lowered as compilation proceeds. Still, the basic structure resembles that 1957 Fortran compiler. It runs a fixed series of passes in a predetermined order.

Modern compilers differ in the set of transformations that they implement, in the algorithms that they use for each transformation, in the order of application for those transformations, and in the languages and machines that they target. We have built an adaptive compiler that discovers, for each input program, an optimization sequence that will minimize a user-selected objective function applied to the compiled code. In this paper, we describe the prototype system and present results from an experiment that tries, for the first time, to characterize the search spaces in which the prototype compiler operates. This knowledge, in turn, will allow us to design better search and steering algorithms for the adaptive compiler. We expect that it will lead to practical applications for this style of search-based optimization.

The next section introduces some of the problems that arise in picking transformations. Section 3 describes our prototype system. Section 4 presents preliminary results from a large-scale experiment to characterize the space of optimizer configurations. Finally, Section 5 lays out some of the issues

*This work has been supported by the Department of Energy through the Los Alamos Computer Science Institute.
Corresponding author: cooper@rice.edu

Algorithm	Scope	Basis	Other Effects
DVNT	*regional*	values	Identities, constants
AWZ	*global*	values	
GCSE	*global*	names	
LCM	*global*	names	Code motion

Table 1: Algorithms for Redundancy Elimination

that must be addressed before this technology can be used in practical compilers.

2. CHOOSING TRANSFORMATIONS

In the last forty-five years, the compiler community has developed hundreds, if not thousands, of transformations. No compiler can implement them all. In designing and building a compiler on the classic model, the compiler writer must choose a specific set of transformations to implement.

To make this task harder, for any given inefficiency that the compiler might target, the compiler writer will find multiple algorithms that attack the problem. These algorithms often catch different cases of the problem and produce different results. To understand the issues, consider the problem of eliminating redundant computations—just one of many such problems that an optimizing compiler must attack. Four algorithms that perform redundancy elimination are shown in Table 1

- Dominator value numbering (DVNT) extends local value numbering [8] to cover acyclic subgraphs of the control-flow graph. It uses hashing to build, bottom-up, a model of the values computed in the region. It replaces redundant expressions with references to earlier results. Along the way, it folds constants and uses algebraic identities to simplify the code [5].
- The Alpern-Wegman-Zadeck algorithm (AWZ) uses a variant of Hopcroft's DFA minimization algorithm to partition the set of expressions in the program into congruence classes [1]. Two expressions in the same class are redundant. A subsequent pass transforms the code to avoid recomputing equivalent expressions.
- Global common subexpression elimination based on available expressions (GCSE) uses data-flow analysis to find redundant expressions [7]. A subsequent pass over the code replaces redundant computations with references to previously computed results.
- Lazy code motion (LCM) extends GCSE to handle expressions that are redundant on some, but not all paths. It inserts evaluations to make these expressions redundant on all paths. It finds optimal placements for these inserted expressions and guarantees that it lengthens no path through the code [18].

While there are many other algorithms for finding redundant expressions and removing them, these four suffice to show the variety that exists in both coverage and results.

- DVNT can discover that `x+x` = `2*x`. The others cannot.
- DVNT and AWZ can prove that `x*y` and `x*z` have the same value when `y` = `z`. Neither GCSE nor LCM can find this, because they base their analysis on names rather than values.
- DVNT cannot find a redundancy that involves control-flow along a back edge in the CFG. All the others can.
- AWZ is optimistic; it finds some redundancies in loops that the others cannot.
- LCM finds partial redundancies. The others do not.

Both DVNT and LCM have additional effects beyond redundancy elimination. A compiler that implements them might avoid the need for other techniques to capture those effects.

Since none of these methods dominates the others, it is not clear which one the compiler should use. Each has strengths. Each has weaknesses. Each catches a different set of cases that might arise in a program.

The best choice depends on the specific set of cases that actually occur in the input program and the impact that improving each occurrence has on overall code quality. To complicate matters further, the passes that precede redundancy elimination in the code will each rewrite the code, creating some opportunities and destroying others. Thus, the best choice may depend on both the input code and the specific passes that run earlier in the compiler. In the same way, the choice for redundancy elimination may affect the choices for later transformations. For the same reasons, running the same set of transformations in different orders can produce different results.

For the remainder of the paper, we assume that each transformation is implemented in an independent, self-contained pass. Each pass contains all the analysis needed to support the transformation. Following this model, we expect that the passes can be run in any arbitrary order.[1]

3. ADAPTIVE COMPILERS

To explore the impact of choice and order on the quality of code produced by a compiler, we have built a prototype adaptive compiler, shown in Figure 2 [11]. Like the 1957 Fortran system, it has a front end, a middle end, and a back end. In addition, it has an explicit, external objective function and a steering algorithm. Its middle end is not the ordered linear sequence of passes found in a traditional compiler; instead, the middle end has a pool of transformations that run in arbitrary orders, as chosen by the steering algorithm.

The steering algorithm uses a series of experiments to find a configuration, or *compilation sequence*, that minimizes the objective function's value for a given input program. The steering algorithm selects a set of initial configurations and directs the compiler to apply those sequences to the input

[1]Some transformations make strong assumptions that restrict reordering. For example, our implementation of partial redundancy elimination requires a name space with specific properties. To handle this, we package the transformation together with a pass that transforms the IR program into the requisite form [4].

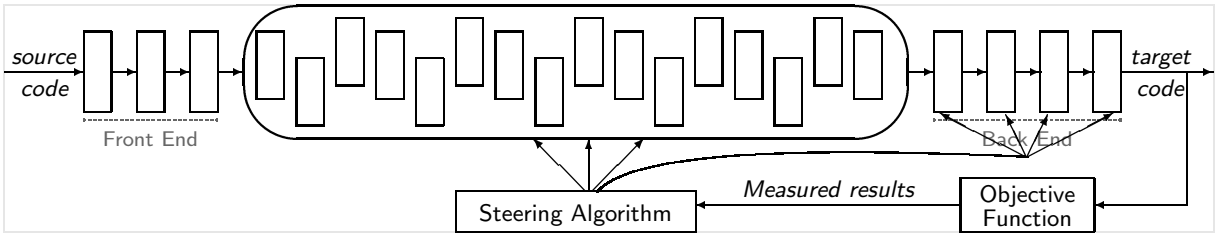


Figure 2: Prototype Adaptive Compiler

program. It uses the compiled code as input to the objective function and evaluates the objective function to obtain fitness scores. The steering algorithm uses both the fitness scores and its historical record to construct the next set of experiments.

Because it uses an explicit objective function, the adaptive compiler can attempt to optimize any measured property of the compiled code. To optimize for program size, the objective function can use the size of the output code as its fitness value. To optimize for speed, the objective function can either run the code and measure the running time or use some performance estimator to compute a fitness value. Choosing the objective function changes the compiler's behavior, to the extent that its pool of transformations can have an effect on the measured property of the output prorgam.

In some cases, the choice of transformations changes a program property as an indirect effect. In one experiment, we used the adaptive compiler to minimize the estimated inter-operation bit transitions, a property that some authors have related to power consumption [20, 28]. The transformations in our prototype do not address bit-transitions directly; instead, the measured bit-transitions are determined by the pseudo-random effects of scheduling and register allocation on the optimized code. However, because different optimization sequences produce different code that feeds into the back end, they can produce different results in the bit-transition metric. In fact, we measured a 6% reduction in inter-operation bit-transitions with the adaptive compiler, comparing against the fixed-sequence version of the same compiler. The adaptive compiler was able to optimize for the impact that the transformations have on the pseudo-random behavior of the back end.

Our prototype adaptive compiler includes a suite of fifteen transformations. They can be run in, essentially, any order (with the exception already noted for partial redundancy elimination). To date, we have built three steering algorithms: a hill-climber, a parameterized genetic algorithm, and an exhaustive search. The current system has three objective functions: run-time cycles, code size, and inter-operation bit-transitions.

3.1 Steering Algorithms

We have implemented three steering algorithms to date.

Hill-climber: The hill-climbing search starts with a single random configuration and systematically improves it. At each step, the search tries to improve the current configuration. We have experimented with a steepest ascent strategy and a randomized strategy. The former finds the largest improvement from the current configuration by systematically testing the options. It halts when no single replacement improves the current configuration. The latter randomly picks a pass to change and a replacement for it. Because it is harder to detect a minimum in this scheme, it halts after a specified number of trials.

Genetic algorithms: The genetic algorithm in the prototype is parameterized to allow experimentation. Our best results, in terms of both final outcome and trials to reach that outcome, have used pools of 100 to 300 variable-length chromosomes,[2] a two-point randomized crossover, and scaled fitness values as weights in making reproductive choice.

One strategy in the genetic algorithm has proven particularly important. When it generates a new chromosome from crossover, it checks for that sequence in the existing pool. If the new sequence is a duplicate, it mutates the new sequence until it is unique. (Since we are using the genetic algorithm to search the configuration space, duplicates are wasted effort.)

Exhaustive enumeration: To explore specific configuration spaces, we have built a steering algorithm that exhaustively enumerates a subspace. This simple algorithm turns the adaptive system into a data collection device. It returns configurations paired with their fitness values from the objective function. We have used this tool in our study with FMIN, described below.

3.2 Objective Functions

We have tested the steering algorithms with three objective functions, as well as combinations of these three.

Speed: This objective function executes the code on a simulator with a given set of input data and returns the total cycle count for the execution. This serves as a proxy for actual execution speed.

Space: This objective function returns the static operation

[2]With fixed-length chromosomes, the genetic algorithm "discovers" NOPs to pad the strings. This necessitates a regimen of "knock-out" testing to find the substring that actually causes the improvement. Because of overlap in optimization coverage, the knock-out test must consider the effects of removing single passes and combinations of passes. It tries all the tests in both forward and backward order over the chromosome. With variable-length chromosomes and tie-breaking in favor of the shorter sequence, these NOPs don't appear in the winning sequences.

count for the generated code. It tracks the code space required by the executable.

Inter-operation bit-transitions: This objective function models a program characteristic that affects the power consumed by the microprocessor in executing the code [20, 28, 17]. The function statically estimates execution frequencies and uses that to approximate bit transitions.

3.3 Engineering the Transformations

For the prototype to work, the transformations that it uses must work when run in arbitrary orders and combinations. Each transformation is a separate pass of the compiler. They consume and produce ILOC, the compiler's definitive IR. All memory allocation is done using an arena-style allocator; this simplifies deallocation and ensures that all such memory is freed [16]. Each pass performs its own analysis. They share code to implement these analyses, but they do not share the results of compile-time analysis—except when that information can be recorded directly in the IR.

This implementation style simplifies the design, implementation, and maintenance of the individual passes. They can be developed, tested, and debugged independently. Once a transformation functions in isolation, we test it in context by inserting it into the compiler at various points. After running literally millions of compilations with the adaptive compiler, we have confidence that these implementations are order independent.

Along the way, we have developed simple fault isolation tools that take a compiler configuration and an input program and discover the minimal set of optimizations that exhibit the fault. We also have a tool that takes a configuration and removes any single passes or pairs of passes that have no effect on the fitness function's value. These tools have proved invaluable.

The current prototype system includes: assertion generation, logical peephole optimization, SCC-based value numbering [29], loop peeling, global constant propagation [30], algebraic reassociation [4], dead code elimination [12], copy coalescing [6], partition-based value numbering [1], strength reduction [10], partial redundancy elimination [21], local value numbering, global renaming, lazy code motion [18], and useless control-flow elimination.

4. EXPERIMENTAL RESULTS

Our prototype adaptive compiler is an experimental tool. In other publications, we have described experimental results that show how the adaptive compiler can produce better code than a fixed sequence compiler for space and for speed [9, 10]. To summarize those results:

- For space, the adaptive compiler produced reductions of 13.5% on average [9], where a direct approach using procedure abstraction based on suffix trees, in the fixed-sequence version of the same compiler, produced reductions of only 5% on average. Furthermore, since the adaptive compiler broke ties in favor of speed, it produced code that was marginally faster than the code from the fixed sequence compiler. In contrast, procedure abstraction always produces code that is marginally slower.
- For speed, the adaptive compiler produced average improvements of 20% percent over the fixed sequence compiler [27]. By breaking ties in favor of smaller code, the adaptive compiler also produced code that was typically smaller than the fixed sequence compiler's code.

In this paper, we report the results of a significant experiment aimed at understanding and improving the behavior of the adaptive compiler and its steering algorithms. The questions we seek to answer are

1. Given a pool of optimizations and an objective function (possibly multi-valued) for a program, what is the number and distribution of sequences that minimize that function for the program? Such information, particularly for local minima, will help us design appropriate steering algorithms. If, for instance, 90% of the sequences in the space minimize the chosen objective function, repeated random walks through the space will suffice to find a good solution.
2. What fraction of the local minima of the objective function are within 10% of the global minimum? How are these "good" minima distributed in the space? That is, what is the probability that a good minimum can be found by a search algorithm guided purely by the local (discrete) gradient of the objective function? Further, what is the effective diameter of the space of optimizations? By that we mean, how many steps will a local search algorithm need on average before settling into a local minimum?
3. Is there a decision rule that can distinguish sequences that lead to a good local minimum from ones that do not? Can such a decision rule be learned from the sampled sequences? Can we relate the decision rule to properties of the program being optimized?

To illustrate our methodology for answering these questions, we will answer them in the context of a optimizing one particular benchmark program, `fmin`. While the particular results that follow pertain to `fmin`, we want to emphasize that this methodology can be applied to any program of interest. **These results are a preliminary step; our real goal is to discover enough about the structure of these spaces so that we can design steering algorithms that can discover excellent compilation sequences from small samples of the configuration space.**

4.1 The FMIN Experiments

We chose `fmin` because we observed that it displayed interesting and complex behavior, even though it is just 150 lines of Fortran [10]. Despite its size, `fmin` has a rather complex control-flow graph—forty-four basic blocks.

The complete configuration space for our adaptive compiler is too large to enumerate at current speeds. We typically run it with a set of fifteen transformations and allow it to use up to fifteen passes. This configuration space contains

15^{15}, or 437,893,890,380,859,000 possible sequences. If we restrict it to ten passes, the configuration space still contains an impractical 576,650,390,625 distinct sequences. Computationally, we felt that 10,000,000 sequences was an upper bound on what we could reasonably explore. This suggested a configuration of ten passes drawn from five different transformations, for 5^{10} or 9,765,625 sequences.

To identify a promising subspace, we ran the adaptive compiler, with a hill-climbing search, from a large set of randomly-generated starting points. We took the five transformations that occurred most frequently in those "winning" sequences and set up an experiment to enumerate the results of all sequences of ten passes drawn from these five.

To date, we have enumerated 5,597,161 individual configurations at roughly 30,000 experiments per dedicated CPU day. By the time the experiment finishes in December 2001, it will have consumed over eleven CPU months.

The best sequence to date produced code that took 1,002 cycles to execute. So far, we have found only one sequence that achieves this result. The worst sequence takes 3,316 cycles to execute. We have found 8,212 sequences that achieve this result. In contrast, code compiled without optimization takes 1,716 cycles to execute. Using the twelve optimization sequence that was the default in our fixed-sequence compiler for years yields code that runs in 1,122 cycles. (Note that the default sequence includes transformations that are not included in the `fmin` experiment.)

Thus, the five transformations in the `fmin` experiment can improve the code by 42% from the unoptimized code or they can make it 93% slower. The difference between these outcomes is a matter of picking the transformations and the order in which to apply them. Even with this limited transformation repertoire, the adaptive compiler finds sequences that improve on the fixed sequence compiler by 11%. If we add dead code elimination to the sequence that produced 1,002 cycles, the sequence produces code that requires 822 cycles to execute—an improvement of 27% over the fixed sequence compiler and 52% over the unoptimized code. Adding this transformation to the `fmin` experiment, however, would increase the search space from just under ten million sequences to over sixty million sequences.

Distribution of the Solutions: The space explored by the `fmin` experiment contains sequences whose fitness value ranges from 1,002 to 3,316. Before we can improve the steering algorithms, we must understand some of the characteristics of the search space.

Figure 3 shows the number of solutions per fitness value across the configurations that have been tested so far. This picture makes clear the discrete nature of the space in which the adaptive compiler operates. For example, there is only one sequence with a fitness value between 1,152 (15% worse than the best) and 1,202 (20% worse than the best).

The table in Figure 4 summarizes this same data. The central column shows the number of solutions that lie within the specified distance from the best solution. The rightmost column shows that figure expressed as a percentage of the

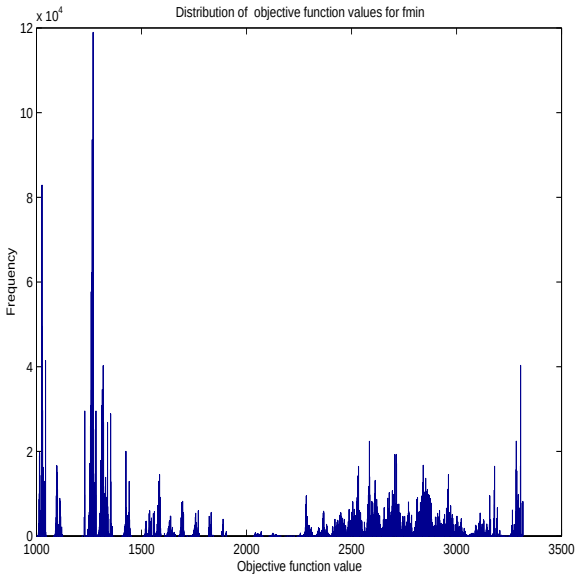


Figure 3: Distribution of `fmin` Solutions

experiments to date (5,597,161). Note that the numbers are cumulative; for example, the one sequence difference between 15% and 20% is precisely the gap seen between 1,152 and 1,202 in Figure 3.

Both Figure 3 and 4 make clear that only 10% of the sequences yield objective function values within 10% of the known global optimum of 1002. A steering algorithm performing a random walk in the space of all sequences will yield a sequence with value within 10% of the global optimum with a probability of 0.1.

Comparison with the Full Transformation Set: If we compile `fmin` with the full set of fifteen transformations, the results are encouraging. Using a genetic algorithm for steering, the adaptive compiler discovered sequences at 822 cycles (in 184 generations of population size 100) and at 825 cycles (in 152 generations of population size 100). Using a hill-climber for its steering algorithm, the adaptive compiler found sequences at 830 cycles (in 750 rounds) and at 833 (in 2,232 rounds). This demonstrates that the potential for improvement exists. Equally important, the adaptive compiler can discover that improvement.

Score	Sequences	% of Solutions
Best	1	0.000018%
1%	14,696	0.26%
2%	91,437	1.6%
5%	480,421	8.6%
10%	576,193	10.3%
15%	640,222	11.4%
20%	640,223	11.4%
25%	708,034	12.6%
Worst	8,212	0.15%

Figure 4: FMIN Solutions Ranked by Quality

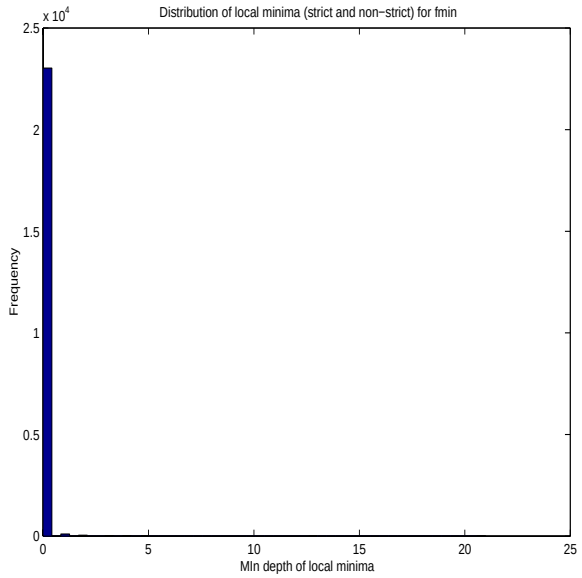


Figure 5: Minimum depth vs. frequency for `fmin`

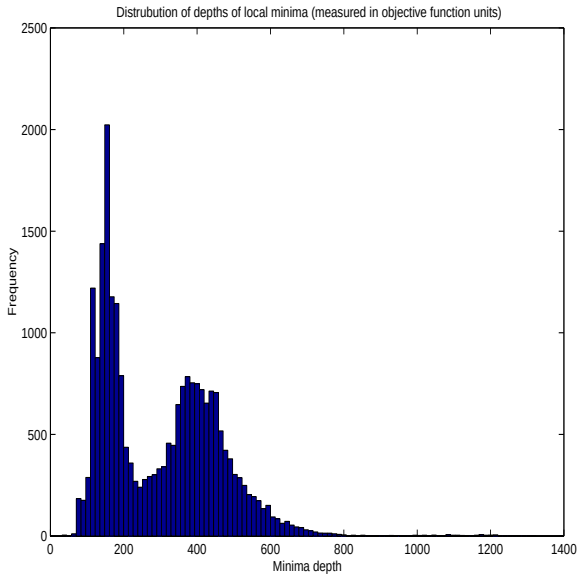


Figure 6: Average depth vs. frequency for `fmin`

4.2 Learning from FMIN

To answer our first question (number and distribution of local minima), we found all sequences with the property that no single-position change produces a better solution. Knowledge of the density of local minima is significant for designing hill climbing steering algorithms that follow the gradient of the objective function. The `fmin` data contains 23,258 local minima; they comprise about 0.42% of the space.

This encouraging news suggests that hill climbing might work well in this space. However, we need to determine if the local minima are strict or not. (A strict local minimum has a value strictly smaller than all of its neighbors.) When there are non-strict local minima, the steering algorithm may need to make several moves on a flat mesa where the objective function value does not change. Figure 5 shows the distribution of depths for each local minima, measured against its lowest neighbor. This illustrates that most local minima are non-strict. This may make it hard to formulate a termination criteria for steepest descent hill climbers.

Figures 6 shows the distribution of depths for each local minima, measured against the average depth of all its neighbors. This chart indicates that, on average, a neighbor of a local minimum is well above it in objective function value. The distribution is bimodal with peaks at roughly 186 and 400. (This is about 18.6% to 40% of the global optimum). The "sides" of each local minimum are, on average, quite steep, which again suggests the value of using hill climbing as the steering algorithm of choice for the problem.

Figures 7, 8 and 9 summarize the performance of a steepest descent hill climber with randomized restarts for the `fmin` problem. The hill climber starts with a random sequence, generates all neighbors involving one change in the sequence (there are 40 possible neighbors in our configuration), and picks the lowest valued among them as the next search point. It stops when there is no change in the objective function value with a unit change in sequence. We ran the hill climber with 100,000 randomly chosen starting points. Figure 7 shows the distribution of the number of steps before the hill climber settled into a local minimum. For a space of over 5 million sequences, the maximum number of hill climbing steps did not exceed 11! The average number of steps is between 5 and 6. The distribution of start values for the objective function is shown in Figure 8; note how closely it resembles the overall distribution of solutions in Figure 3.

Figure 9 shows the final values achieved by the hill climber from these 100,000 random starts. It demonstrates the effectiveness of steepest descent hill climbing for this problem. 94.97% of the final solutions are within 10% of the global minimum. This means that we can expect the hill climber to find one of these solutons with a probability of 0.9497.

Figure 10 shows that there is no correlation between the number of steps taken by the hill climber and the choice of starting point, while Figure 11 makes the same observation from the point of view of ending point of the search. This shows that the height of the initial starting point does not offer any clues as to the number of steps needed to reach a local minimum. There as many good solutions located a few steps away from a random initial starting point as there are solutions far away.

This analysis has helped answer the first two questions raised at the start of the section. Such an analysis can be used to study any program of interest. However, the daunting cost of data collection to help perform such an analysis raises doubts about the practicality of the technique. We now present evidence to show that sparse random sampling of the entire space of over 5 million sequences is sufficient to build predictive models of performance. In particular, we have used three kinds of performance models: two based on classification learning which discriminate sequences that yields solutions within 10% of the global optimum from

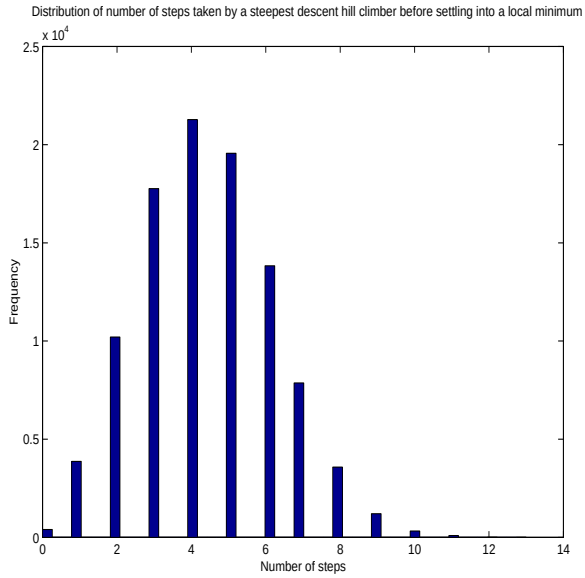


Figure 7: Distribution of number of steps taken by a steepest descent hill climber starting from a random sequence before settling into a local minimum

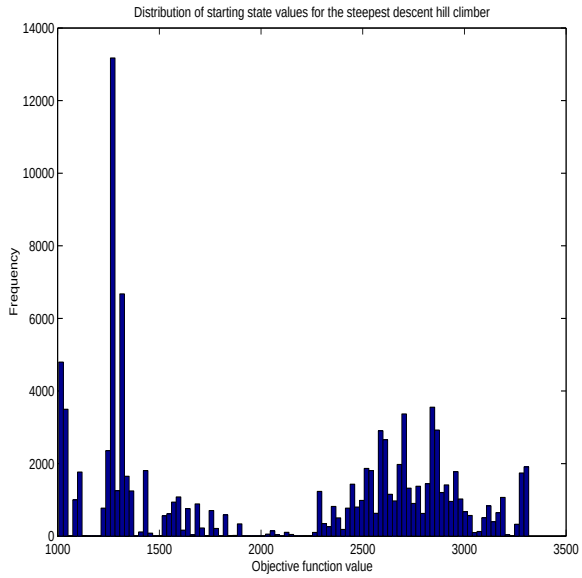


Figure 8: The distribution of initial objective function values for the steepest descent hillclimber when started from 100,000 randomly chosen start states

those that do not, and a probabilistic model of the successful and unsuccessful sequences that separates the two categories cleanly. The number of samples of the underlying sequence space needed to robustly learn these models is 10,000 (about 0.2% of the entire space).

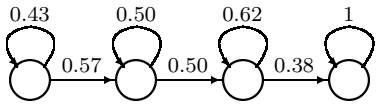
The two classification technqiues used were decision trees and support vector machines. For both these techniques, labeled sequences are provided and the goal is learn a function that discriminated "good" from "bad" sequences. 5000 good and 5000 bad sequences were sufficient to learn compact decision trees with error rates of 3.9% (trained in a leave-out-test-set mode). The decision tree identifies positions in the sequence and the optimization needed at that position to make a sequence good. Support vector machines were trained on the problem and they yielded predictive models with accuracies of 94% with a polynomial kernel of degree 4 [**?**]. The use of polynomial kernels of degree 4 suggests that interactions between 4 of the 10 positions in the sequence shape whether or not a sequence is good. The decision tree was able to identify these particular interactions.

We used the c4.5 program [**?**] implementing the induction algorithm in [**?**] to construct a decision tree classifier. The tree is derived from 5,000 good and 5,000 bad sequences from `fmin`. A decision tree is a representation of a disjunction of conjunctions of elementary expressions of the form (position i in sequence = one of the five available optimizations). The number of leaves in the decision tree is the number of disjunctions, and the depth of the tree shows the maximum number of conjuncts in each disjunction. Our learned decision tree contains 96 leaves and has a depth of 4. 37 of the leaves encode the description of the function that identifies good sequences. One of these disjuncts is (position 2=0 and position 3=3 and position 10=4). This disjunct demonstrates that there are long range dependencies in the data between positions 2, 3 and 10 in the optimization sequence.

The decision tree has an estimated error of 3.9% on unseen examples, making it an accurate model of sequence costs.

Even for a program as small as fmin it is clear that we could not have analytically discovered these interactions. However, we have demonstrated that a methodology based on sparse sampling coupled with inductive machine learning can build models of interactions between optimizations for particular programs. The algorithms are fast enough and their sample requirements modest enough to make them part of a pre-compilation analysis for importnat programs.

To understand how probabilistic models will work on these problems, we trained two hidden markov models (HMMs): one for good sequences and the other for bad sequences [**?**] An HMM is a Markov chain where each state generates an output. Only the outputs are observable, and the goal is to infer the underlying state sequence. HMMs are useful for modeling of sequences and time-series data, since the discrete state-space of the Markov chain can be used to capture long-range dependencies in the underlying data. Given sequences that yield solutions to within 10% of the optimum, we use expectation maximization [**?**] to learn a four state left-to-right HMM that generates those sequences.



0	1	0.57	0.25	0.3
1	0	0	0.009	0.157
2	0	0	0.25	0.147
3	0	0.08	0.195	0.106
4	0	0.35	0.292	0.287

Hierachical Markov Model for Good Sequences

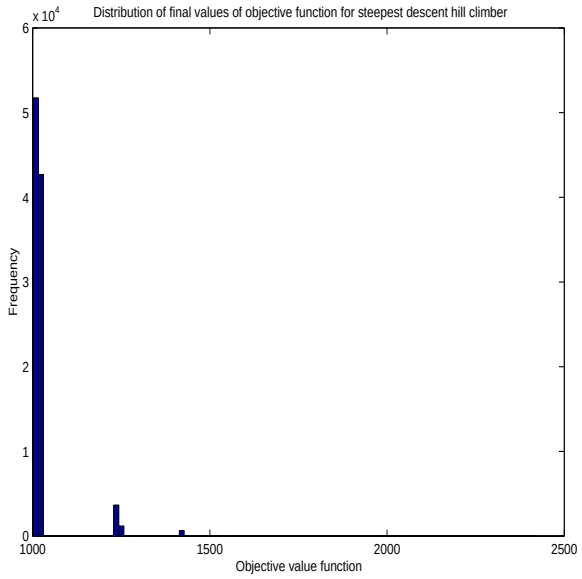


Figure 9: The effectiveness of the steepest descent hillclimber for `fmin`. The distribution of final values obtained by the hill climber when started from 100,000 randomly chosen start states.

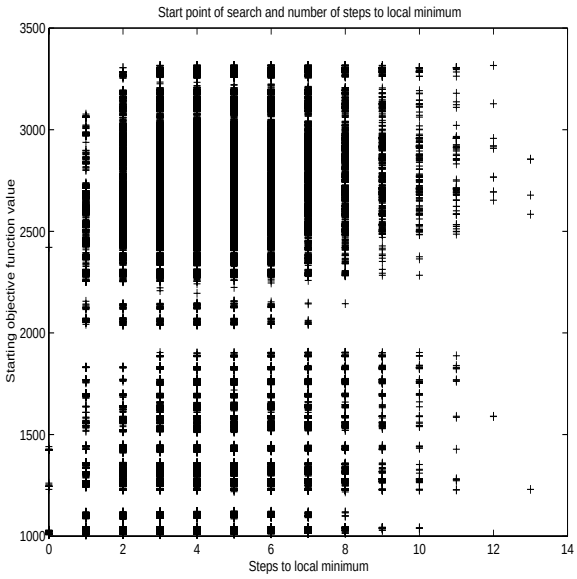
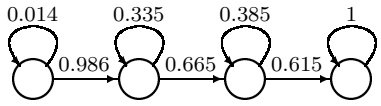


Figure 10: The relationship between the number of steps needed by steepest descent hill climber to reach a local minimum and the objective function value at the randomly picked start state, for `fmin`.

The HMM can be treated as a compact stochastic description of all the example sequences. The HMM parameters appear to be remarkably robust for HMMs learned from the good as well as bad sequences.[3] The following HMM was generated for the bad sequences:



0	1	0.18	0	0.25
1	0	0.22	0.21	0.17
2	0	0.6	0.41	0.19
3	0	0.0	0.16	0.22
4	0	0	0.26	0.17

Hierachical Markov Model for Bad Sequences

The remarkable fact is how different these two HMMs are. This shows that good and bad sequences can be stochastically discriminated with great precision for `fmin`.

5. OPEN QUESTIONS

Our work with adaptive compilation has raised as many questions as it has answered.

5.1 Stopping Criteria

A significant practical problem with the both the genetic algorithm and the hill-climbing search is the difficulty that they have recognizing success. To make these techniques practical, we need a way to decide that the current solution is likely to be within ϵ of optimal, for reasonable values of ϵ.

A major motivation for experiments like the `fmin` experiment is the hope that we can discover enough about the search space to let the algorithm recognize good local minima. Characterizations of the depth of local minima, both in terms of objective function values and in terms of the number of moves required to escape, may prove helpful. For example, if we know that most local minima require a three-position replacement to escape, then the algorithm might discontinue a search (and restart) when it discovers that three-sequence replacements are not generating better fitness values.

5.2 Proxies and Estimators

The adaptive compiler works, effectively, by conducting a series of small experiments and using the results to guide its search for a configuration that minimizes the objective function. The speed of these experiments is a serious limit on our ability to use the adaptive search. When it finishes in December, 2001, the `fmin` experiment will have consumed under one CPU year.

To speed up these experiments, we are investigating the use of proxies and estimators for evaluating objective functions. For example, what is the impact on final code quality of using static estimates of run-time behavior? Can we develop simple models for memory accesses and computation that are close enough for the steering algorithm? Good estimators and proxies may go a long way toward making these ideas practical for routine use—even if they are only used in the initial stages of the search.

[3]We used the same sequences to learn the decision tree, the support vector machines and the HMMs.

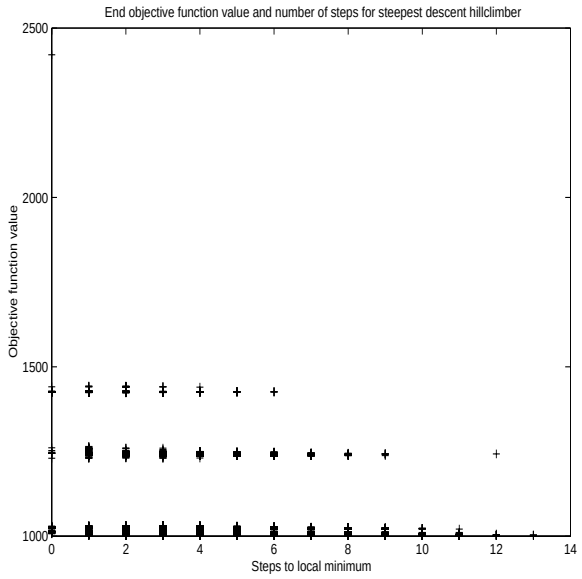


Figure 11: The relationship between the number of steps needed by steepest descent hill climber to reach a local minimum and the objective function value at the final local minimum state, for `fmin`.

6. RELATED PRIOR WORK

Early work on adaptive compilers [22] have focused on feeding dynamic profile information from program execution back into the compiler to guide optimization. Other attempts to use search in optimization include Nisbet's system, which used genetic algorithms in an attempt to parallelize loop nests [23, 13], and Massalin's Superoptimizer, which used exhaustive search in an attempt to perform optimal instruction selection [19]. Nisbet's system was ineffective, probably because search was not a good fit to his problem. Massalin's technique produced good results, but was too expensive for routine use. Granlund and Kenner adapted Massalin's ideas to produce a design-time tool that generates assembly sequences for use in GCC's code generator [14].

Schielke used iterative repair with random restart to study instruction scheduling [27]; his work shows the value of using large-scale, computationally-intense studies to derive knowledge about hard problems in compilation.

Our preliminary work using a genetic algorithm to find compilation sequences suggests that search is a good fit to the problem and that adaptive randomized sampling can yield good results in a reasonable amount of time [9, 11].

A few authors have written on the interactions between optimizations in a specific way. In the context of performing incremental global optimization, Pollock and Soffa encountered such interaction; they characterized the interactions among the set of transformations in their system [25, 26]. Whitfield and Soffa characterized the interactions between a set of optimizations, including their ability to enable and disable opportunities for each other [31]. Later, they designed a systematic way of describing interactions between optimizations and for analyzing and working with those interactions [32]. Padua *et al.* have studied the impact of different transformations on the ability of their Polaris compiler to parallelize loops [3, 24].

Acknowledgements

This work would be impossible without the many tools built by members of the Massively Scalar Compiler Project over the years. Phil Schielke got us started on this work with his curiousity. Andrew MacKay and L. Alamagor built various implementations of the search algorithms. To all of these people go our heartfelt thanks.

7. REFERENCES

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, CA, USA, Jan. 1988.

[2] 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. 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. Institute of Radio Engineers, NY, NY, USA, Feb. 1957.

[3] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996.

[4] B. E. Boser, I. M. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, ACM Press:144–152, 1992.

[5] P. Briggs and K. D. Cooper. Effective partial redundancy elimination. *SIGPLAN Notices*, 29(6):159–170, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[6] P. Briggs, K. D. Cooper, and L. T. Simpson. Value numbering. *Software–Practice and Experience*, 27(6):701–724, June 1997.

[7] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–456, May 1994.

[8] J. Cocke. Global common subexpression elimination. *SIGPLAN Notices*, 5(7):20–24, July 1970. In *Proceedings of a Symposium on Compiler Construction.*

[9] J. Cocke and J. T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

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

[11] K. D. Cooper, L. T. Simpson, and C. A. Vick. Operator strength reduction. *ACM Transactions on Programming Languages and Systems*, 2001. *to appear.*

[12] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21^{st} century. In *Proceedings of the 2001 LACSI Symposium.* Los Alamos Computer Science Institute, Oct. 2001. Available at `http://www.cs.rice.edu/ keith/LACSI`.

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, Oct. 1991.

[14] N. G. Fournier. Enhancement of an evolutionary optimising compiler. Master's thesis, Department of Computer Science, University of Manchester, Sept. 1999.

[15] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the gnu C compiler. *SIGPLAN Notices*, 27(7):341–352, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[16] Suif. group. Documentation accompanying the suif system. Available from `http://suif.cs.stanford.edu`.

[17] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software—Practice and Experience*, 20(1):5–12, Jan. 1990.

[18] M. Kandemir, N. Vijaykrishnan, M. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Proceedings of the Internation Symposium on Computer Architecture*, June 2000.

[19] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. *SIGPLAN Notices*, 27(7):224–234, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[20] H. Massalin. Superoptimizer – A look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Systems*, pages 122–126, Palo Alto, CA, USA, 1987.

[21] H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh. Techniques for low energy software. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 1997.

[22] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, Feb. 1979.

[23] T. Night. Web site for the Transit Project. See: `www.ai.mit.edu/projects/transit/tn101/tn101.html`.

[24] A. P. Nisbet. GAPS: A compiler framework for genetic algorithm (GA) optimised parallelisation. In *Poster Session at the International Conference and Exhibition on High-Performance Computing and Networking, HPCN Europe '98*, 1998.

[25] Y. Paek and D. Padua. Experimental study of compiler techniques for NUMA machines. In *Proceedings of the IEEE International Parallel Processing Symposium and Symposium on Paralle and Distributed Processing*, Apr. 1998.

[26] L. L. Pollock. *An Approach to Incremental Compilation of Optimized Code.* PhD thesis, University of Pittsburgh, Department of Computer Science, 1986.

[27] L. L. Pollock and M. L. Soffa. Incremental global reoptimization of programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(2):173–200, Apr. 1992.

[28] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(81-106), 1986.

[29] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA, 1993.

[30] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proc. of the IEEE*, 77(2):257:286, February 1989.

[31] P. J. Schielke. *Stochastic Instruction Scheduling.* PhD thesis, Rice University, Department of Computer Science, May 2000.

[32] D. Shin and J. Kim. An operation rearrangement technique for low-power VLIW instruction fetch. In *Proceedings of the Workshop on Complexity-Effective Design*, June 2000.

[33] L. T. Simpson. *Value-Driven Redundancy Elimination.* PhD thesis, Rice University, 1996.

[34] M. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr. 1991.

[35] D. L. Whitfield and M. L. Soffa. An approach to ordering optimizing transformations. *SIGPLAN Notices*, 25(3):137–146, Mar. 1990. Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP).

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