

# Exploring the Structure of the Space of Compilation Sequences Using Randomized Search Algorithms<sup>†</sup>

Keith D. Cooper Alexander Grosul Timothy J. Harvey Steve Reeves  
Devika Subramanian Linda Torczon Todd Waterman

*Rice University  
Houston, Texas, USA*

## Abstract

Modern optimizing compilers apply a fixed sequence of optimizations, which we call a *compilation sequence*, to each program that they compile. These compilers let the user modify their behavior in a small number of specified ways, using command-line flags (*e.g.*, `-O1`, `-O2`, ...). For five years, we have been working with compilers that automatically select an appropriate compilation sequence for each input program. These *adaptive compilers* discover a good compilation sequence tailored to the input program, the target machine, and a user-chosen objective function. We have shown, as have others, that program-specific sequences can produce better results than any single universal sequence [1, 23, 7, 10, 21]

Our adaptive compiler looks for compilation sequences in a large and complex search space. Its typical compilation sequence includes 10 passes (with possible repeats) chosen from the 16 available—there are  $16^{10}$  or 1,099,511,627,776 such sequences. To learn about the properties of such spaces, we have studied subspaces that consist of 10 passes drawn from a set of 5 ( $5^{10}$  or 9,765,625 sequences). These 10-of-5 subspaces are small enough that we can analyze them thoroughly but large enough to reflect important properties of the full spaces. This paper reports, in detail, on our analysis of several of these subspaces and on the consequences of those observed properties for the design of search algorithms.

## 1 Compilation Sequences

Compilers operate by applying a fixed sequence of optimizations, called a compilation sequence, to all programs. The compiler writer must select ten to twenty optimizations from the hundreds that have been pro-

posed in the literature; then the compiler writer must select an order in which they should execute. Choosing the right optimizations and the right order for one specific program is hard. The compiler writer must choose a set that works well for all programs.

The compiler writer must choose a limited set of techniques to include in the default compilation sequence. A given optimization only improves programs exhibiting the inefficiencies that it targets. For one program, the problem of optimization choice can be solved: pick techniques that address that code’s inefficiencies. The compiler’s “universal” sequence, however, must work well for all programs. Thus, these sequences tend to include optimizations that are broadly applicable rather than high-payoff techniques with a more narrow focus.<sup>1</sup> As Robison observes: “Compile-time program optimizations are similar to poetry: more are written than are actually published in commercial compilers” [19].

The compiler writer must also pick an order in which to execute the optimizations. We have little theoretical understanding of the effect of a given compilation sequence on the properties of the compiled code that it produces. The interactions and interdependences between optimizations are complex and uncharacterized. Transformation *a* may create opportunities for later application of *b*; alternately, it may eliminate opportunities for another transformation *c* [18, 23, 6]. In fact, this behavior is also program specific; *a*’s ability to create or eliminate opportunities depends on the presence of specific features in the code being compiled.

To address these problems—compilation choice and compilation order—we have developed a new compiler structure. Our *adaptive compiler* uses a program-

<sup>†</sup> This work has been supported by the Los Alamos Computer Science Institute and by the National Science Foundation through grant CCR-0205303.

<sup>1</sup>In practice, every compiler has economic limits: compile time, developer effort, calendar time before release. These constraints limit the number of optimizations that will be implemented. In this constrained environment, more general techniques are the safe strategy.

specific compilation sequence to optimize each program. Our work has shown that program-specific compilation sequences can produce better code than can a “universal” sequence [8, 1]. Today, our prototype compiler must find these program-specific sequences with search techniques. To design effective search techniques, we must understand the properties of the spaces that the compiler searches.

Without a theory that accounts for the impact of individual transformations and the interactions between several of them in sequence,<sup>2</sup> the compiler cannot compute a fitness function or a quality measure for a proposed sequence except by evaluating it: compiling the program and measuring properties of the compiled code. Further, we cannot determine how good a sequence is compared to a sequence that optimizes a given objective function—because we cannot identify the optimal sequence except with exhaustive search. The spaces are too large to examine each solution. Our goal is to develop search algorithms that, with high probability, find “good” solutions: solutions with near-optimal fitness values. If we can make the search for such sequences reasonably efficient, then discovering and using program-specific compilation sequences will be practical.

Our approach has been to use empirical techniques to discover properties of these search spaces. We enumerated subspaces, recorded the results, and analyzed that data to discover properties of those spaces that might affect the behavior of search algorithms. To confirm that those properties carry over into larger search spaces and into spaces for other programs, we have conducted exploratory experiments with a larger set of benchmarks and a larger set of optimizations. Many of the observed properties of the subspaces also appear to hold in these other spaces.

We have shown that program-specific sequences produce consistently better results than any universal sequence [1]. This paper reports on our preliminary analysis of several 10-of-5 subspaces. Specifically, it examines the following questions:

1. What percentage of the sequences have fitness values within a given distance of optimal?
2. How are these “good” sequences distributed in the search spaces?
3. Are local minima<sup>3</sup> numerous? How are their fitness values distributed?

<sup>2</sup>Others, notably Soffa *et al.*, are working on models that capture these effects [24].

<sup>3</sup>The family of search algorithms we consider are randomized hill climbing algorithms, and the distribution and preponderance of local minima in the search space determine their overall efficacy.

Equally important, we need to understand the structural properties of the search spaces. Do search techniques outperform random sampling? Can searches capitalize on the structure of the spaces?

The remainder of this paper is organized as follows. Section 2 describes our approach and provides necessary background detail on the methods used in the enumeration study. Section 3 presents the results of our initial exploratory surveys of two 10-of-5 subspaces. Section 4 describes the results of our experiments that attempted to confirm for a 10-of-16 space and for larger benchmarks some of properties observed in the subspace studies. Section 5 presents one of the first empirically derived cost-benefit trade-off curves for program-specific compilation sequences. These curves demonstrate the dominance of our search techniques over random sampling in spaces where the probability of finding a good sequence is low, and where there is structure in the form of long downhill runs in fitness values in the space.

## 2 Experimental Plan

The experiments described in this paper consist of two phases. In the first phase, we selected two interesting but small Fortran programs and a set of 5 optimizations taken from the 16 in our prototype compiler (see Table 1). We enumerated the corresponding 10-of-5 space for each program. This is the set of  $5^{10}$  sequences, each of length ten, constructed from the five chosen optimizations. Note that repeated passes are considered. We compiled the code for an abstract RISC machine and simulated its execution to determine the number of operations executed. We analyzed the data, offline, to discover answers to the questions raised in Section 1.

In the second phase, we applied the insights gained in the first phase to searches in a larger 10-of-16 space, using a larger set of programs. We evaluated three techniques, an impatient hill climber, a genetic algorithm, and random sampling. Our results suggest that many of the properties observed in the enumeration studies carry over into these larger spaces.

*Using an Abstract Machine* The work in this paper focuses on searching the space of compilation sequences. The details of evaluation matter much less than the match between program and optimizations and the interactions among optimizations. Using a simulated abstract machine as a target allows us to create an instrumented, homogeneous execution environment using a variety of target machines. The many experiments that underlie this paper took many CPU months. Because we used the simulated abstract machine, we were able to perform the experiments on a

p	<i>loop peeling</i> peels the first iteration of each innermost loop
l	<i>partial redundancy elimination</i> finds and eliminates redundancies and partial redundancies [17]
o	<i>peephole optimization</i> examines logically adjacent operations and tries to simplify them [13]
s	register coalescing eliminates register-to-register copy operations [5]
n	<i>useless control-flow elimination</i> eliminates empty blocks and redundant control-flow [11]
c	<i>sparse conditional constant propagation</i> combines optimistic constant propagation with unreachable code elimination [22]
d	<i>dead code elimination</i> based on SSA-form [12, 11]
g	<i>optimistic value numbering</i> uses partitioning to find global redundancies [2]
m	<i>renaming</i> builds a name space suitable for the implementations of either <code>l</code> or <code>z</code> . The compiler inserts it automatically before <code>l</code> or <code>z</code> .
r	<i>algebraic reassociation</i> uses commutative and distributive law to reorder expressions [3]
t	<i>strength reduction</i> replaces iterated multiplies with iterated additions [9]
u	<i>local value numbering</i> folds constants and eliminates redundancies [11]
v	<i>SCC value numbering</i> implements an optimistic, global version of value numbering [20]
x	<i>DVNT</i> performs value numbering over dominator trees [4]
y	<i>EBB value numbering</i> performs value numbering over extended basic blocks [4]
z	<i>lazy code motion</i> improves on <code>l</code> with more careful placement of inserted operations [16]

Table 1: Passes available in the prototype compiler

variety of machines. We can also compare and contrast the results with those of experiments that we do in the future—after the current hardware is gone.<sup>4</sup>

**Background on the Enumerations** The two programs, `fmin` and `zeroin`, are both taken from Forsythe, Malcolm, and Moler’s classic book on numerical algorithms [14]. `fmin` computes the minimum of a unimodal function by a combination of golden section search and parabolic interpolation. `zeroin` searches for the zero of an input function between given bounds. Both programs are small (see Table 2 in Section 4).

`fmin` has a complex control-flow graph [10]. The column labeled `ILOC` lines in Table 2 gives the number of operations when the code is translated into `ILOC`,

<sup>4</sup>To validate that the ideas and results hold for other machines, we have also performed a series of experiments using our SPARC code generator. The results on both the 10-of-5 and 10-of-16 spaces are similar to those described in Section 4.

the instructions for the simulated abstract machine referred to earlier, that we use as an intermediate code in the compiler [11].

To select 5 transformations for the enumerations, we ran a hill-climbing algorithm to find good sequences in a 10-of-16 space. The full set of transformations is shown in Table 1; they address scalar inefficiencies rather than memory performance [15, 11]. The objective function was `ILOC` operations executed, which we also refer to as the dynamic operation count. We started the hill climber from 100 randomly chosen points and recorded the final sequence of each hill-climber run. Finally, we computed the frequency with which each transformation appears in the winning sequences. We chose the top 5 transformations selected by this process: `p`, `l`, `o`, `s`, and `n`, listed at the top of Table 1.

We exhaustively enumerated the 10-of-5 space for both `fmin` and `zeroin` using these 5 transformations, designated `fmin+plosn` and `zeroin+plosn`. The initial enumeration of `fmin+plosn` required 14 CPU-months and 6 calendar-months on a set of 3 machines. Subsequent improvements to the testing harness and the compiler’s basic infrastructure have improved our testing speed; today, we can enumerate `fmin+plosn` in less than 40 CPU-days on a collection of SUN and MacIntosh workstations. The enumeration scales well because it consists of millions of independent tasks.

### 3 Surveying the Landscape

As a starting point, we enumerated the `fmin+plosn` and `zeroin+plosn` spaces, using dynamic operation counts as the fitness value. Figure 1 shows the distribution of dynamic operation counts for the two spaces. Note that both distributions are discrete, with large gaps in the fitness values. For instance, there are no sequences with operation counts that lie between 14% and 20% of the optimum. The best sequence in `fmin+plosn` is 41.6% better than the empty compilation sequence (1002 vs 1716 operations), and the best sequence in `zeroin+plosn` is 42.5% better (832 vs 1446 operations). No sequence in either space does worse than the empty compilation sequence. In the full space, however, it is easy to get results that are worse than the empty sequence, by a factor of two or more. The bottom graph in Figure 1 shows cumulative distributions for both spaces. Roughly 20% of the sequences lie within 10% of the optimum, while nearly 30% of the sequences fall within 20% of the optimum. These statistics suggest that good sequences are common in both spaces.

If these solutions are distributed uniformly, we would expect repeated random sampling of the space

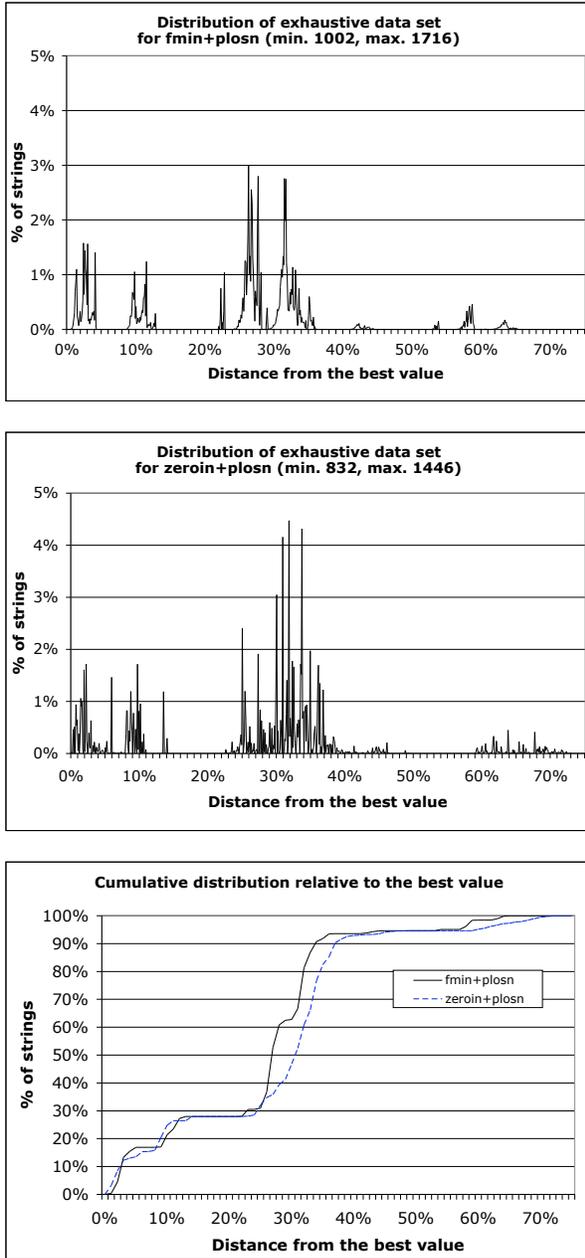


Figure 1: The distribution of dynamic operation counts for `fmin+plosn` (top) and `zeroin+plosn` (center). The bottom plot shows the percentage of sequences that are within  $x\%$  of the optimal solution for `fmin+plosn` and for `zeroin+plosn`.

to yield good sequences. Let a good sequence be one whose performance is within  $x\%$  of the optimum, and let  $p$  be the probability that a random sequence in the subspace is good. The probability of picking a good sequence at the  $N^{\text{th}}$  probe, after failing the previous  $N - 1$  times is  $p(1 - p)^{N-1}$ . If  $\epsilon$  is a bound on the probability of failure to obtain a good sequence after

$N$  independent random probes, we have

$$p + p(1 - p) + p(1 - p)^2 + \dots + p(1 - p)^{N-1} \geq 1 - \epsilon$$

which can be used to solve for the number of random samples,  $N$ , needed to guarantee a solution within  $x\%$  of the optimum with probability greater than  $1 - \epsilon$ . Simple algebra yields:

$$N \leq \frac{\log \epsilon}{\log(1 - p)}$$

Note that number of samples needed ( $N$ ) depends only on the percentage  $p$  of good solutions in the space, and the probability  $1 - \epsilon$  of not missing a good solution. It is independent of the size of the space! For `fmin+plosn` and `zeroin+plosn`, if we want solutions that are within 10% of the optimum, with probability greater than 0.999, we need no more than 43 independent random samples.

The bound on  $N$  above assumes a uniform distribution of good sequences throughout the space. Does this assumption hold for `fmin+plosn` and `zeroin+plosn`? To get a handle on this question, we must empirically estimate the distribution. The estimation requires that we define a neighborhood relation among sequences. We say that sequence  $a$  is a neighbor of sequence  $b$  if they differ in exactly one position—that is, their Hamming distance is one. Other possible definitions of neighborhood remain to be investigated. The results presented in this paper define  $a$ 's neighbors as the set of strings at Hamming-1 distance.

To examine the distribution of solutions, we begin by building a graph whose nodes are sequences with fitness values that lie within  $x\%$  of the optimum. We add an edge between any two nodes whose sequences are at Hamming-1 distance. Any connected component of this graph is considered a *cluster*. We consider a singleton node, connected to no other node, as a degenerate cluster. Figure 2 shows the number and sizes of the clusters in the sequence space as a function of  $x$ , the percentage difference in the fitness value of a sequence from that of an optimal sequence. For  $x < 2.6\%$ , there are multiple isolated clusters (as many as 42, most of them of size 1), and a single large cluster. The bottom plot shows this effect well. Further, it appears that the single large cluster is quite well separated in the sequence space from the smaller, isolated clusters for  $x < 2.6\%$ . The average Hamming distance between sequences in the large cluster and those in the smaller clusters is 6.5, while the Hamming distance between the sequences in smaller clusters themselves averages 2.8.

A dramatic transition occurs in the clustering of `fmin+plosn` sequences with dynamic operation counts greater than 2.6% of the optimal sequence. At that

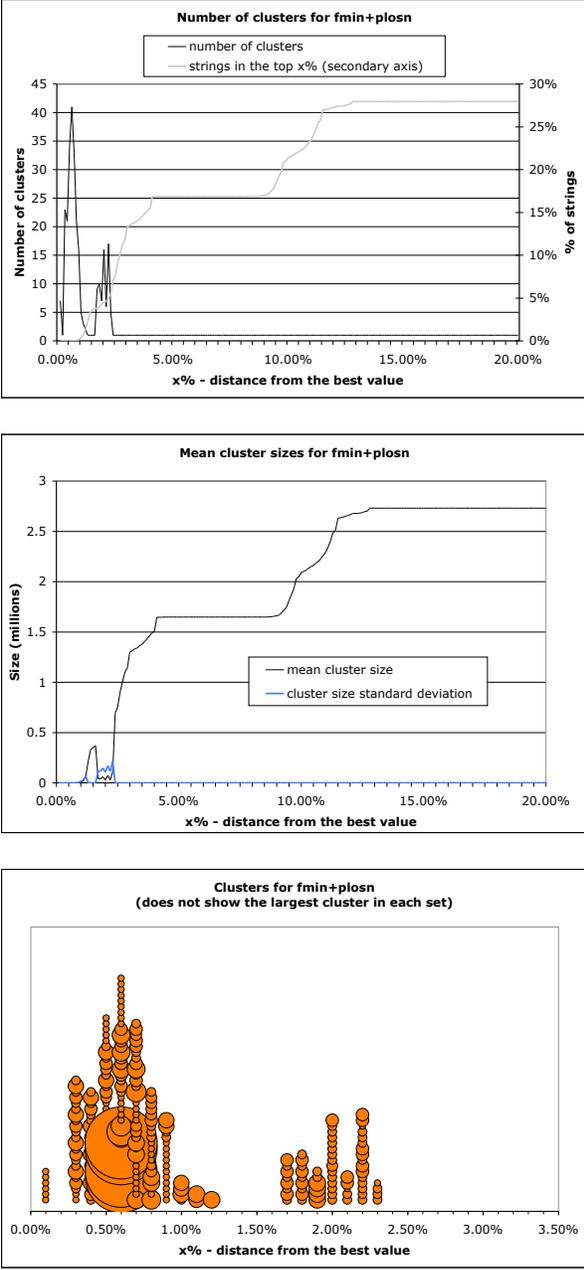


Figure 2: The top plot shows the number of clusters in the `fmin+plosn` as a function of solution quality, measured as percentage from the optimal. Note that the number of clusters drops to 1 at  $x = 2.6\%$ . The center plot shows variation in cluster sizes as a function of solution quality. The bottom plot is a scale representation of how clusters (other than the single giant cluster) vary with solution quality. The plots for `zeroin+plosn` are similar.

point, all sequences coalesce into a single cluster. The number of good sequences also rises sharply. This change suggests a phase transition at  $x = 2.6\%$  in the average case complexity of finding a good sequence in `fmin+plosn`. The existence of the phase

transition implies that the task of finding a sequence that is within 2.6% of the optimum in `fmin+plosn`, is qualitatively different from the task of finding sequences with a looser quality bound ( $x > 2.6\%$ ). For the “hard” region, an algorithm with multiple restarts is important to ensure that it can reach isolated clusters. For the “easy” region, simple random sampling might suffice, due to the large number of good sequences and their widespread distribution in the spaces.

While cluster analysis provides an overall view of the distribution of good solutions in the spaces, it gives no insight into the difficulty of reaching good solutions from different starting points. Does the terrain have a natural slope that can guide a descent algorithm to a good solution? How widespread are local minima in the space? How many of these local minima are “good” (i.e., within  $x\%$  of the optimum)? `fmin+plosn` contains 31,995 local minima, of which 189 are *strict*—each Hamming-1 neighbor has a higher fitness value. The top graph in Figure 3 shows the distribution of local minima for `fmin+plosn`. This is a bimodal distribution. Note the logarithmic scale on the  $y$  axis, which is the number of local minima. For  $x \leq 2.0\%$ , less than 13% of the local minima are good. Between 2.0% and 2.6% the number of good local minima rises sharply from 13% to 80%. For  $x$  between 2.6, and 4%, 80% of the local minima are good. Again, we see a phase transition in the complexity of the problem around 2.6%. Below 2.6%, few local minima are good; above 2.6%, most local minima are good.

To further understand the structure of the sequence space, in particular, how the local minima are distributed, we examined the behavior of several descent algorithms on both the `fmin+plosn` as well as the `zeroin+plosn` spaces. To take a step, the algorithms generate a series of Hamming-1 neighbors of the current sequence and find their fitness values. In the 10-of-5 space there are 40 Hamming-1 neighbors for every sequence. The algorithms generate the neighbors in *random* order and take the *first* downhill step that they discover. Some of the algorithms are *impatient*; they examine a limited percentage of a sequence’s neighbors before declaring the current sequence to be a local minimum. The center plot of Figure 3 shows the probability of being able to make a downhill step (i.e., finding a lower neighbor) as a function of the number of neighbors generated by an impatient descent algorithm. At a patience level of 10% (i.e., no more than 4 random neighbors are examined at each step), the probability of being able to continue with the descent is 76.59%. That is, 23.41% of the time, the descent stops prematurely at the 10%

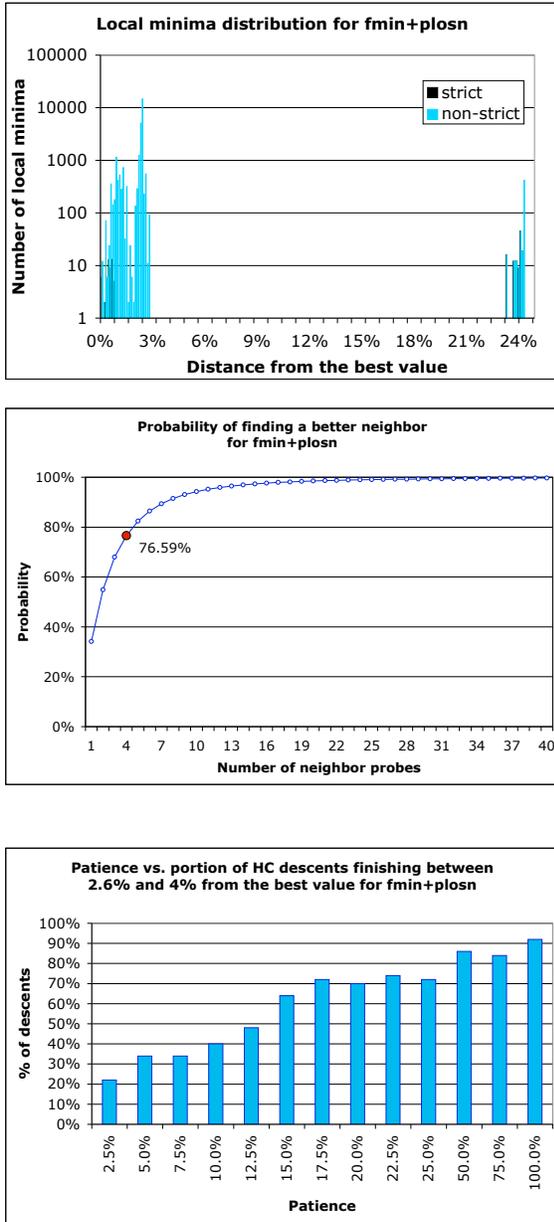


Figure 3: The top plot shows the distribution of local minima in  $fmin+plosn$ . The center plot shows the probability of finding a descent step as a function of the number of neighbors examines. 4 neighbors is equivalent to 10% patience, since the number of neighbors of a sequence in a 10-of-5 space is 40. The bottom plot shows the probability of an individual descent run terminating in a local minimum between 2.6 to 4 percent of the optimal sequence, as a function of patience.

patience level. If more than half the neighbors are examined, the probability of finding a downhill step comes close to 1. These are the probabilities for each individual descent step.

How do we select patience levels for descent algo-

gorithms? To do this in a principled manner, we need to know how likely an algorithm with a certain patience level is to achieve a local minimum between 2.6% and 4% of the optimal value at termination. We call these minima, “good” minima. The reason we focus on this range of local minima is that they comprise more than 80% of all local minima. The minima whose values are below 2.6% of the optimum are very difficult to reach, unless we have a starting sequence very close (between 2 to 4 steps) to those minima. The bottom plot in Figure 3 presents the variation with patience of the probability of termination at a good local minimum. At the 10% patience level, the probability of an individual descent run terminating in a good local minimum is 40%, while at the 20% patience level, this probability rises to 70%. A descent algorithm with patience level of 20% examines more than twice the number of sequences in each individual run as one with patience of 10% because it makes longer descent runs in the space. Patient descent algorithms which examine all 40 neighbors of a sequence take on average between six to seven steps. This amounts to between 240 to 280 evaluation per restart. In contrast, across both  $fmin+plosn$  and  $zeroin+plosn$ , the average number of descent steps taken at patience level of 10% is 1.9. This amounts to about 8 evaluations per restart. For the 20% patience level, we have 8 neighbors times about 4 descents per run, which is 32 evaluations for each restart. In this space, termination in a local minimum between 2.6% to 4% of the optimal requires a starting point that is close (within 6 descent steps) to that local minimum. Given a fixed number of evaluations, it is better to keep the number of evaluations per restart low, and to maximize the number of restarts. A patience level of 10% allows four times the number of restarts as a patience level of 20%. The low cost per impatient-descent run makes multiple randomized restarts with 10% patience value, a very attractive strategy. Multiple restarts from random initial sequences increase the likelihood of finding a good local minimum in this space.

As a final confirmation of the difficult nature of these spaces, consider the plot of the 4-of-5 subspace of  $fmin+plosn$ , shown in Figure 4. Fitness values are plotted as a function of the sequence’s prefix and suffix, each of which is two characters long. The space shows many sharp peaks and valleys, with no obvious global minimum. The lack of long descent paths in this space, and the preponderance of local minima are made obvious in this visualization.

*Implications for Search Design* The craggy nature of these spaces as shown in Figure 4, with many sharply

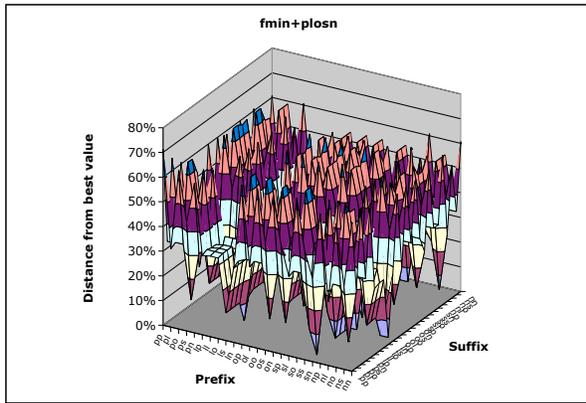


Figure 4: Surface plot of 4-of-5 subspace, **fmin+plosn**. Instead of plotting the  $5^4$  sequences on a single axis, we split each sequence into a two character prefix and a two character suffix from the alphabet **plosn**. Each grid point on the  $xy$  plane represents a sequence of length 4, and the  $z$ -axis of the plot shows its fitness value.

defined local minima, makes descent algorithms a natural choice. The key considerations must be efficiency and likelihood of finding a good solution.

Impatience is a cost-benefit tradeoff. It lets a descent algorithm bound the number of sequences that it must evaluate. It creates the situation where the algorithm may halt before reaching a true local minimum. The middle plot in Figure 3 suggests that three-quarters of the time, or more, a patience level of 10% will suffice to find a descending step. Assuming that these properties carry over from the enumerated subspaces into larger spaces, using a 10% threshold for patience, especially when combined with repeated randomized restarts, should efficiently produce good results.

The cluster analysis of **fmin+plosn** shows (and **zeroin+plosn** confirms) that the algorithm should easily find solutions between 2.6% and 4% of the optimum. Finding solutions within 2.6% of the optimum is much harder, as only 13% of the local minima fall in that range. A strategy of multiple trials from random starting points will let a descent algorithm find a “good” local minima. Given the data suggesting that a descent algorithm with patience level of 10% is fast—that is, it takes relatively few steps and examines relatively few neighbors per step—multiple randomized restarts of an descent algorithm at the 10% patience level should find good solutions well.

The data in Figure 5 bears out these conjectures. The top plot shows the number of restarts needed for an impatient descent algorithm (with patience level 10%), as a function of solution quality, in **fmin+plosn**. Note the dramatic drop after 2%, which again reflects the phase change that we saw in the earlier

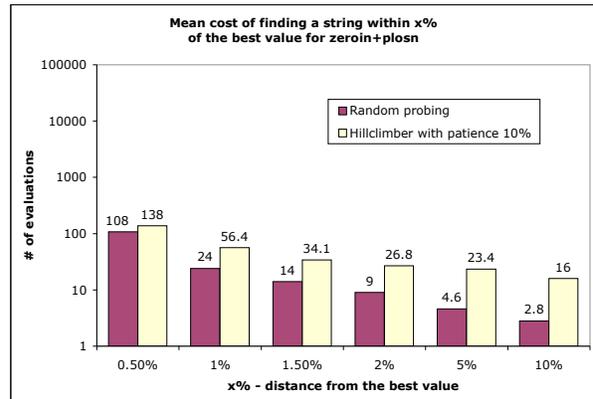
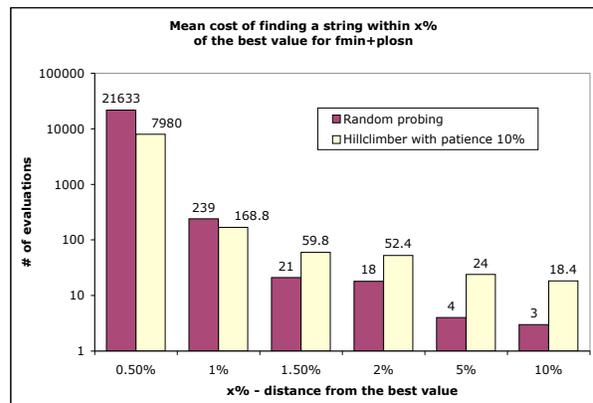
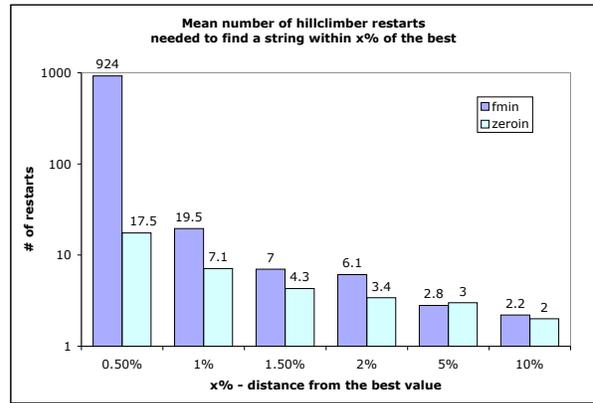


Figure 5: Comparing randomized restart impatient descent algorithm versus independent random sampling for **fmin+plosn** and **zeroin+plosn**.

data at 2.6%. (It requires 924 restarts to find a solution within 0.5% of optimal, 6.1 to find one within 2% and 2.8 restarts to find one within 5%. The data for **zeroin** follows a lower-valued but similarly shaped curve.) The lower plots compare the average number of evaluations required by the impatient descent algorithm (here labeled as a hill climber) with patience level of 10%, against random probing of the space,

in `fmin+plosn` and in `zeroin+plosn`. For solutions within 1% or less of optimum, the impatient descent algorithm wins. As solution quality decreases, random probing becomes the more effective method.

*Section Summary* This survey of the landscape of `fmin+plosn` and `zeroin+plosn` offers one of the first computational glimpses of the space of compilation sequences. Any analytical theory for sequence design or prediction must capture the shape of the surface shown in Figure 4. Our empirical analysis of the enumerated subspaces shows that there are many local minima in the space, and that randomized restarts should be an effective strategy to overcome shallow local minima. The terrain is very rough and most descent runs are short. Starting points matter for local search algorithms; deep local minima are reachable only if the starting points are very close to those minima. There is a sharp phase transition in this space. The problem of finding solutions that are within 2.6% of the optimum is qualitatively harder than the problem of finding solutions that are greater than 2.6% of the optimum. These properties need to be taken into account in the design of effective search algorithms for this space.

## 4 Exploring the Full Space

The complete mappings of `fmin+plosn` and `zeroin+plosn` are only useful to the extent that they provide insights into searching the larger compilation-sequence spaces that will arise in realistic contexts. We conjecture that the broad-brush characteristics of these small spaces also hold for larger spaces and other programs. In particular, we expect that:

- the larger spaces will have many shallow local minima;
- the probability of a random sequence being close to optimal (within 1%) is low; and
- the complexity of finding solutions within  $x\%$  of optimal will exhibit phase transitions for certain values of  $x$ .

Unfortunately, the spaces of real interest are too large to enumerate, so we cannot know the global minimum fitness value. This fundamental lack of knowledge makes it difficult to compare our solutions to an optimal one. To analyze this larger set of benchmarks in the larger 10-of-16 space, we choose to compare the results against those obtained with our compiler’s default compilation sequence.

To verify these expectations, we conducted a series of experiments using an expanded benchmark set (Table 2) and the full set of transformations (Table 1) to derive 10-of-16 sequences. We used two different

Name	Suite	# of Proc’s	Source Lines	ILOC ops.	# of Blocks
<code>adpcm-c</code>	Media	1	192	479	37
<code>adpcm-d</code>	Media	1	168	424	30
<code>g721-c</code>	Media	16	965	4066	268
<code>g721-d</code>	Media	21	1080	4879	344
<code>tomcatv</code>	SPEC	1	192	2599	78
<code>svd</code>	FMM	1	351	2493	185
<code>zeroin</code>	FMM	1	125	332	39
<code>fmin</code>	FMM	1	160	434	59

Table 2: A list of the benchmark programs studied in this paper. “Media” indicates the MediaBench suite; “Spec” indicates the SPEC benchmark suite; and FMM refers to the Forsythe, Malcolm, and Moler library [14]. Programs above the line are written in C; those below the line are written in Fortran 77. # of Blocks refers to the number of basic blocks in the control flow graph of the program.

randomized local search algorithms: a descent algorithm and a genetic algorithm. We compare the results of compilation against the performance of our compiler’s universal sequence, `rvzcodtvzcod`.

The descent algorithm is the impatient algorithm with patience level of 10% described earlier. This implementation runs from 50 randomly-chosen starting points and retains the best result. We call this version HC 50.

The genetic algorithm was derived after extensive experimentation to find good parameter settings. It uses populations of 50 and 100 sequences, a single-point random crossover and fitness proportional selection. It uses 10% elitism (the top 10% of sequences survive without change), and a mutation probability of 0.02. An unusual feature of this GA is its treatment of duplicate sequences. If selection-and-crossover yields a sequence that already has been evaluated by the current run, the GA replaces the duplicate with a randomly-selected, untried sequence. This process eliminates duplicate evaluations of the same sequence in a single GA run. Each GA runs for 100 generations. We call the algorithms GA 50 and GA 100, with population of 50 and 100 sequences.

Table 3 shows the results of using HC 50, GA 50, GA 100, for `fmin` and `zeroin`, along with the results produced by the compiler’s universal sequence. In addition, the table shows the results of random probing for 200 trials (R200) and 2000 trials (R2000). Dynamic operation counts are given as a percentage improvement from the results of the universal sequence (1,136 operations for `fmin` and 978 for `zeroin`). Since these algorithms are randomized, we report the mean and standard deviations on dynamic operation counts over three independent runs.

	fmin					g721-c			
	DynOps	StDev	Cost	Sequence		DynOps	StDev	Cost	Sequence
<i>universal</i>	<b>1,136</b>		1	rvzcodtvzcod	<i>universal</i>	<b>426.2</b>		1	rvzcodtvzcod
GA 50	26.5%	0.35	4,550	pppxocdlsn	GA 50	18.7%	0.81	4,550	pnpppcdzsn
GA 100	26.5%	0.15	9,110	opzpdppxsn	GA 100	19.2%	0.16	9,110	pppppcdvsn
HC 50 10%	26.1%	0.13	2,124	noppplxdsn	HC 50 10%	16.7%	0.67	2,857	nzpppncpds
R200	14.3%	0.56	200	zodyvvtgos	R200	7.7%	2.15	200	pzucvnsuss
R2000	17.0%	3.90	2,000	nzyypoogvsd	R2000	12.4%	0.08	2,000	cxsdxvgcsn

	zeroin					g721-d			
	DynOps	StDev	Cost	Sequence		DynOps	StDev	Cost	Sequence
<i>universal</i>	<b>978</b>		1	rvzcodtvzcod	<i>universal</i>	<b>779.5</b>		1	rvzcodtvzcod
GA 50	30.4%	0.18	4,550	oplvscdzsn	GA 50	19.2%	1.42	4,550	vpppppcdzsn
GA 100	30.4%	0.18	9,110	pozvscvdsn	GA 100	19.5%	0.57	9,110	pppppcdvsn
HC 50 10%	29.0%	0.56	2,054	zodnpvodnsn	HC 50 10%	17.9%	1.20	2,752	tvppppscnd
R200	16.0%	2.13	200	zodyvvtgos	R200	7.7%	1.99	200	pzucvnsuss
R2000	20.0%	2.04	2,000	nzyypoogvsd	R2000	12.5%	0.22	2,000	cxsdxvgcsn

Table 3: The performance of HC 50, GA 50, GA 100, R200 and R2000 against the universal sequence for `fmin` and `zeroin`, shown as percentage improvement. Note the diversity in the sequences; inter-sequence Hamming distance is as large as 8.

Table 5: The performance of GA 50, GA 100, HC 50, R200 and R2000 against the universal sequence for `g721-c` and for `g721-d`. The dynamic operation counts are in millions. Inter-sequence Hamming distance varies between 6 and 8.

	adpcm-c			
	DynOps	StDev	Cost	Sequence
<i>universal</i>	<b>13.3</b>		1	rvzcodtvzcod
GA 50	33.0%	0.56	4,550	prppocvdsn
GA 100	33.0%	0.56	9,110	rpzpodcvsn
HC 50 10%	31.5%	0.36	2,238	nppcnlgpds
R200	24.0%	2.47	200	pzucvnsuss
R2000	29.0%	0.32	2,000	cxsdxvgcsn

	adpcm-d			
	DynOps	StDev	Cost	Sequence
<i>universal</i>	<b>11.1</b>		1	rvzcodtvzcod
GA 50	30.6%	0.01	4,550	prpopcdlsn
GA 100	30.6%	0.00	9,110	rppocdpvsn
HC 50 10%	30.0%	0.01	2,174	yrxrsvpscn
R200	24.5%	3.37	200	nulltrmzsz
R2000	30.0%	0.00	2,000	xscrndgvsn

Table 4: The performance of HC 50, GA 50, GA 100, R200 and R2000 against the universal sequence for `adpcm-coder` and `adpcm-decoder`. The dynamic operation counts are in the millions. Inter-sequence Hamming distance varies from 4 to 8.

HC 50, GA 50 and GA 100 all find sequences that are 26% better than the universal sequence for both `fmin` and `zeroin`. They also outperform random sampling; however, the two GA versions use significantly more evaluations than R2000. In contrast, HC 50 examines an average of 2100 sequences for both these programs. In both solution quality and work, HC 50 dominates random sampling, which suggests that the probability of finding a good solution

at random is very low.

As a final point, note the diversity of the derived sequences. This behavior suggests that the 10-of-16 spaces contain isolated clusters of equivalent good sequences, as we saw in `fmin+plon` for sequences that fall within 2.6% of optimal.

Table 4 shows results for the five algorithms on `adpcm-coder` and `adpcm-decoder`. Again, HC 50, GA 50, and GA 100 find better sequences—between 30% and 33% better than the universal sequence. HC 50 and R2000 are similar in solution quality and effort. While the final sequences still show diversity, the Hamming distance between them is lower than in Table 3. This property suggests that good sequences are distributed among a smaller number of clusters in the 10-of-16 spaces for `adpcm-coder` and `adpcm-decoder` than in the 10-of-16 spaces for `fmin` and `zeroin`. The fact that R2000 performs nearly as well as HC 50 for the same effort suggests that the 10-of-16 spaces for `adpcm-coder` and `adpcm-decoder` contain more good sequences than the spaces for `fmin` and `zeroin`.<sup>5</sup>

Table 5 shows our results for `g721-coder` and `g721-decoder`. HC 50, GA 50 and GA 100 all find solutions that are 16% to 19% better than the universal sequence. Again, the derived sequences show large diversity, with inter-sequence Hamming distances in the range of 6 to 8. The Hamming distances suggest

<sup>5</sup>It would be interesting to determine whether there are solutions that are more than 32% better than the universal sequence, and how they are distributed in the space.

	tomcatv			
	DynOps	StDev	Cost	Sequence
<i>universal</i>	<b>220.0</b>		1	rvzcodtvzcod
GA 50	17.3%	1.36	4,550	crxpotvdsn
GA 100	17.3%	1.31	9,110	rxpcotdvsn
HC 50 10%	17.0%	2.69	2,555	rytlxosnsd
R200	0.3%	5.32	200	lrvyldodn
R2000	7.1%	1.82	2,000	ycrozssddd

Table 6: The performance of GA 50, GA 100, HC 50, R200 and R2000 against the universal sequence for `tomcatv`. Inter-sequence Hamming distance varies between 4 and 8.

	svd			
	DynOps	StDev	Cost	Sequence
<i>universal</i>	<b>5,120</b>		1	rvzcodtvzcod
GA 50	26.1%	0.07	4,550	cztpvodvsn
GA 100	30.1%	2.34	9,110	ocdvtpvdsn
HC 50 10%	24.2%	1.84	2,756	potpvsdnpn
R200	1.8%	4.43	200	lrvyldodn
R2000	11.7%	0.74	2,000	nzyoopgvsd

Table 7: The performance of GA 50, GA 100 HC 50, R200 and R2000 against the universal sequence for `svd`. The different sequences suggest isolated clusters of equivalent solutions. Hamming distance between sequences varies from 4 to 8.

the presence of isolated clusters in the solution space. The poor performance of R200 and R2000 demonstrates that the probability of a random sequence in this space being better than the universal sequence is extremely low.

Table 6 shows the results for `tomcatv` from the SPEC benchmarks. HC 50, GA 50 and GA 100 find solutions that are 17% better than the universal sequence. The random sampling techniques R200 and R2000 fail to find good solutions, suggesting that the probability of stumbling into a good solution in this space is also extremely low.

Table 7 shows the results for the `svd` program. HC 50 finds improvements of 24% over the universal sequence. The inter-sequence Hamming distance between solutions found in three independent runs of HC 50 suggests that there are isolated clusters of equivalent solutions. GA 50 and GA 100 produce better solutions, improving on the universal sequence by 26% to 30%. The inter-sequence Hamming distance for the GA solutions is also large, ranging from 4 to 8. The sparsity of good solutions and their distribution in isolated clusters is confirmed by the poor performance of R200 and R 2000.

**Section Summary** The randomized local search algorithms, HC 50, GA 50, and GA 100 find solutions that improve by 15% to 30% over the compiler’s uni-

versal sequence. Contrasting the behavior of R200 and R2000 against the others provides insight into the difficulty of finding good solutions, while the inter-sequence Hamming distances of the solutions provides some insight into the clustering of solutions in the various spaces.

## 5 The Economics of Sequence Design

Figure 6 summarizes the discussion in the previous section and presents the cost/benefit tradeoff for finding program-specific compilation sequences for the benchmarks examined in the previous section. Our benchmarks cleanly divide into two categories: those for which impatient descent algorithms with randomized restarts outperform simple random sampling, and those for which random sampling is competitive with descent algorithms. For `adpcm-coder` and `adpcm-decoder`, there are a large number of good sequences in the 10-of-16 space. There is little structure in the space of solutions; the average number of descents in each run of HC 50 is small. For such spaces, the cost of descent is not worth the expected payoff. Random sampling with 2000 probes strikes the right cost/benefit tradeoff for finding good compilation sequences in these spaces. For `fmin`, `zeroin`, `g721-encoder`, `g721-decoder`, `tomcatv` and `svd`, HC 50 dominates the other algorithms. Neither GA algorithm is cost-effective because they both yield solutions that are similar in quality to HC 50 for two to five times the effort expended by HC 50. For these benchmarks, good solutions are sparse, and are located in isolated clusters. There is some structure in the solution spaces, as evidenced by the somewhat longer descent runs of HC 50 compared to those in the `adpcm` benchmarks. However, relative to the diameter of these sequence spaces, these descent runs are still short, and the overall topography of the space is similar to the one shown in Figure 4. Random sampling fails to generate solutions of sufficient quality for these benchmarks, because the probability of a random sequence being a good one is extremely low.

## Conclusions

This paper presents empirical data on the structure of the sequence space for several interesting programs. Exhaustive enumerations of `fmin+plosn` and `zeroin+plosn` yield insights into the difficulties of analytically characterizing the impact of sequences on particular programs. The analyses suggest design criteria for search algorithms to work in the full sequence space and to work with larger programs. Our experiments in the 10-of-16 space indicate that lessons from

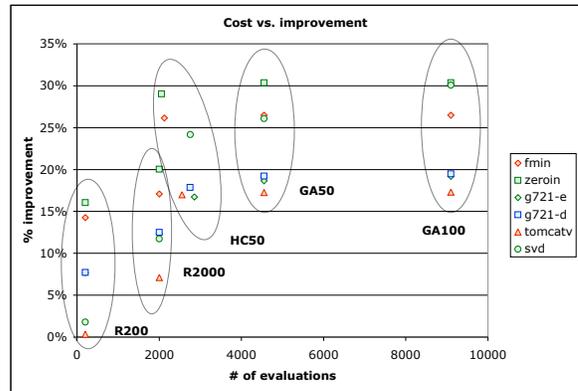
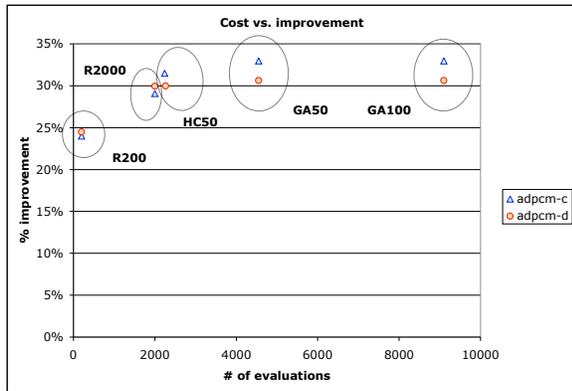


Figure 6: The cost-benefit tradeoff curve for HC 50, GA 50, GA 100, R200, and R2000, in 10-of-16 spaces for the benchmarks studied in this paper. The most cost-effective position is the upper left-hand corners of both figures. Our benchmarks divide into two categories: (adpcm-coder, adpcm-decoder), and (fmin, zeroin, g721-encoder, g721-decoder, tomcatv, svd). For adpcm-coder and adpcm-decoder, R2000 is nearly as effective as HC 50, while for all other benchmarks HC 50 clearly dominates R2000. adpcm-coder and adpcm-decoder are characterized by the abundance of good solutions in the 10-of-16 space. For all other benchmarks, good solutions are sparse, and they are located in isolated clusters in the space. There is also some structure in the solution space, which descent algorithms take advantage of.

the exploratory survey have utility beyond the particular programs we investigated. Finally, Section 5 shows the first estimates of the empirical economic tradeoffs that a compiler must make to decide how to find program-specific compilation sequences.

## Acknowledgements

This work was supported by the National Science Foundation through grant CCR-0205303 and by the Los Alamos Computer Science Institute. The views expressed in this article should not be construed as the official views of either agency. Many people contributed to building the software system that serves as the basis for these experiments. To all these people, we owe a debt of thanks.

## References

- [1] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 231–239, June 2004.
- [2] Bowen Alpern, Mark N. Wegman, and F. Kenneth 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, January 1988.
- [3] Preston Briggs and Keith 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*.
- [4] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Software-Practice and Experience*, 27(6):701–724, June 1997.
- [5] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via graph coloring. *Computer Languages*, 6(1):47–57, January 1981.
- [6] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, 1995.
- [7] Keith D. Cooper, Philip J. Schielke, and Devika 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.
- [8] Keith D. Cooper, Philip J. Schielke, and Devika 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.

- [9] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. Operator strength reduction. *ACM Transactions on Programming Languages and Systems*, 2001. *to appear*.
- [10] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21<sup>st</sup> century. In *Proceedings of the 2001 LACSI Symposium*. Los Alamos Computer Science Institute, October 2001. Available at <http://www.cs.rice.edu/~keith/Adapt>.
- [11] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan-Kaufmann Publishers, 2003.
- [12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, October 1991.
- [13] Jack W. Davidson and Christopher W. Fraser. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(2):191–202, April 1980.
- [14] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler. *Computer Methods for Mathematical Computations*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1977.
- [15] Stefan Goedecker and Adolfo Hoesie. *Performance Optimization of Numerically Intensive Codes*. SIAM, Philadelphia, PA, USA, 2001.
- [16] Jens Knoop, Oliver Rüthing, and Bernhard 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*.
- [17] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.
- [18] Lori L. Pollock. *An Approach to Incremental Compilation of Optimized Code*. PhD thesis, University of Pittsburgh, Department of Computer Science, 1986.
- [19] A. D. Robison. Impact of economics on compiler optimization. In *Proceedings of the ACM 2001 Java Grande Conference, Stanford University*, pages 1–10, 2001.
- [20] L. Taylor Simpson. *Value-Driven Redundancy Elimination*. PhD thesis, Rice University, 1996.
- [21] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, 2003.
- [22] Mark Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.
- [23] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):1053–1084, November 1997.
- [24] M. Zhao, B. Childers, and M.L. Soffa. Predicting the impact of optimizations for embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 1–11, June 2003.