

Searching for Compilation Sequences*

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

*Rice University
Houston, Texas, USA*

Abstract

A growing body of literature on adaptive compilation suggests that using program-specific [7] or function-specific [24] compilation sequences can produce consistent improvements over compiling the same code with a traditional fixed-sequence compiler [18, 1, 27, 24]. The early work on this problem used genetic algorithms (GAs) [7]. GAs find good solutions to these problems. However, they must probe the search space thousands of times; each probe compiles and evaluates the code.

To build a practical compiler that discovers good compilation sequences, we need techniques that find good sequences with much less effort than the GAs require. To find such techniques, we embarked on a detailed study of the search spaces in which the compiler operates. By understanding the properties of these spaces, we can design more effective searches.

This paper focuses on effective search algorithms for the problem of choosing compilation sequences – an ordered list of optimizations to apply to the input program. It summarizes the search-space properties that we discovered in our studies. It presents and evaluates two new search methods, designed with knowledge of the search-space properties. It compares the new methods against the best sequence-finding GA that we have developed.

Our new search methods can find good solutions with 400 to 600 probes. The first GA for sequence finding required 10,000 to 20,000 probes [7]. Our most effective GA runs for 2,300 probes. The strength of these results validates our paradigm – learn about the spaces and use that knowledge to improve the search techniques.

*This work has been supported by the Los Alamos Computer Science Institute and by the National Science Foundation through grant CCR-0205303.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to publish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

1. INTRODUCTION

Several researchers have shown that feedback-driven optimizing compilers can produce better code than is possible with a compiler that always uses the same approach – optimization choice and order as well as heuristics to choose parameters – for all programs [27, 7, 24, 23, 19, 18, 1]. These feedback-driven compilers exhibit fluid behavior; they adjust the optimization sequence or the parameters used by those optimizations, or both, to fit a specific application.

The complex interactions between optimizations makes it difficult to predict or to discover the best set of behaviors for a given compiler and application. Thus, these next-generation compilers use feedback and search to find compiler configurations that produce good results. Since each probe of the search space involves compiling the code and evaluating the resulting code's performance, the efficacy of feedback-driven compilation depends heavily on development of effective search techniques. Prior work has established that a compiler which tries multiple sequences can produce better code than can a compiler that uses a single fixed-sequence [7, 24, 19, 18], and that finding good sequences is hard [1, 6].

Early work in this arena focused on genetic algorithms (GAs) [7, 19] and parameter sweeps [16, 15]. Prior work shows these approaches have found good results; unfortunately, both methods probe the search space many times. Schielke's GA ran for 1,000 generations with a population of size 20; thus, it probed the space up to 20,000 times.¹ Because the GAs lack good stopping criteria, the compiler must run them for a fixed number of generations; likewise, parameter sweeps typically run to completion.

To make adaptive, feedback-driven compilation practical, we must reduce the number of probes required to find a good solution. Triantifyllis *et al.* engineered a practical system by identifying a small set of good sequences and trying each of those sequences on the application [24]. This approach captures some of the benefits of adaptation at a fixed increase in compile time. Our approach differs from theirs; we have studied the search spaces in which our prototype adaptive compiler operates and used the derived knowledge to design effective search techniques.

This paper presents search techniques for the compilation sequence problem. One of these, an impatient random-descent algorithm called HC-10 finds good solutions with 400 to 600 probes of the search space – an order of mag-

¹Hashing the sequences and their results reduced the number of evaluations by about 45% on average, giving rise to our estimate of 10,000 to 20,000 evaluations.

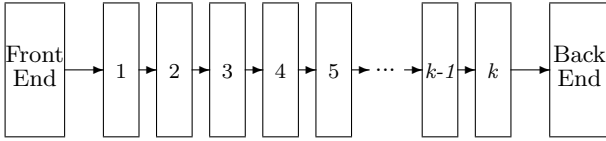


Figure 1: Choosing a Sequence

nitide improvement over the GAs previously used to solve this problem and $4\times$ less effort than *our* best GA. Both HC-10 and the GA employ multiple randomized trials. Both are progressive; with more time the search produces better results. For a search that uses between 250 and 1,000 evaluations, HC-10 appears to be the algorithm of choice. Somewhere between 1,000 and 2,000 evaluations, our best GA becomes a more effective way to find solutions.

The strength of these results validates our paradigm – learn about the spaces through intense exploration, study the properties offline, and design appropriate search techniques. We expect that this paradigm will lead to practical search methods for other hard problems in adaptive compilation, such as selection of sites to inline [25].

2. COMPILATION SEQUENCES

A typical optimizing compiler consists of a front end, an optimizer, and a back end. The front end handles language specific tasks of scanning, parsing, and semantic elaboration; it also produces a representation of the code in some intermediate representation (IR). The back end translates the IR form of the program into the instruction set of the target processor. The back end typically performs instruction selection, scheduling, and register allocation. The optimizer’s task is to transform the IR program so that the back end can generate better code for the target-machine.

The work described in this paper focuses on the structure and operation of the optimizer. We assume that the optimizer is structured as a set of independent passes, each consuming and producing an IR program, as shown in Figure 1. The IR form of the program is definitive; that is, it encodes all the information known about the program. In a traditional compiler, the selection and order of these passes is determined at design time; one sequence or a small number of predetermined sequences are used (perhaps one for `-O1`, another for `-O2`, and so on). Almost no compiler lets the user select an order for the passes in the optimizer. (Our prototype [7, 1] and the VISTA system [29, 19] are rare exceptions.)

Given this compiler structure, the compilation-sequence problem is deceptively simple:

Given a set of transformations, find a sequence of k transformations that produces good executable code for the input program.

To simplify discussion, we assign each transformation a single-character name. Thus, a sequence for $k=10$ becomes a 10-character string. That string may repeat some transformations and ignore others.

The experiments in this paper use strings of length 10 ($k=10$) drawn from a set of 16 transformations, a configuration that we call “10-of-16.”² The transformations, shown

²Schielke’s GA worked in a 12-of-10 space with the same compiler. It used only 10 of the 16 passes available in the current system [7].

in Table 1, can run in any order. The 10-of-16 space includes 16^{10} , or 1,099,511,627,776, sequences. For a given program, we can evaluate any sequence in the space by compiling the code and measuring its speed (by running the code or by estimating its performance). It is impractical, however, to explore more than a small fraction of the sequences in such a space.

Ideally, we could define “good” in terms of a solution’s distance from the global optimum. As we will show, the spaces that arise in practice are complex enough that we cannot find a global minimum analytically. Equally frustrating, the spaces are sufficiently large that we cannot afford to explore them completely to find a global minimum. Thus, we cannot know how far a solution is from optimal. To evaluate solution quality, we will use two different metrics: improvement over a fixed-sequence compiler in the full 10-of-16 space, and measured solution quality in tractable subspaces.

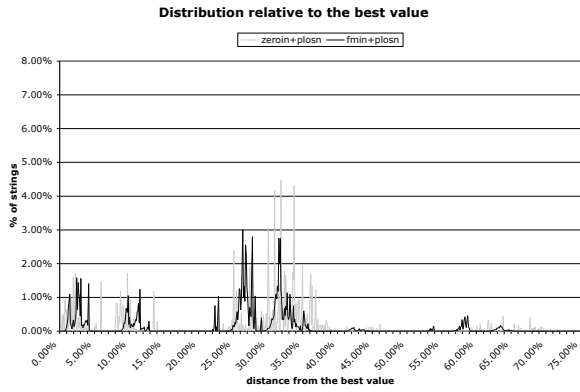
This paper focuses on making sequence-finding compilers practical by drastically reducing the cost of finding a good solution. To achieve this goal, we have designed search algorithms that take into account the observed characteristics of the spaces in which those algorithms will operate. Section 3 summarizes the observed characteristics of the search spaces encountered by our prototype compiler. Section 4 describes three search algorithms that perform well in these spaces. Section 5 presents experimental results that show the effectiveness of these algorithms. Section 6 uses the results from the previous sections to show the relative effectiveness of the various techniques.

3. OBSERVED PROPERTIES

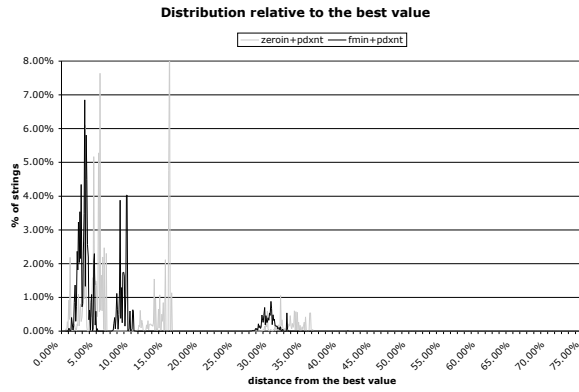
To learn about the properties of the search spaces in which our adaptive compiler operates, we conducted a series of experiments that enumerated 10-of-5 subspaces for small benchmark programs. A 10-of-5 subspace contains 5^{10} , or 9,765,625, sequences. The enumeration compiles and executes a benchmark program with every sequence in the subspace. To choose our initial set of transformations, we ran a hillclimber from 100 randomly-chosen points in the 10-of-16 space, compiling the benchmark program `fmin`. We chose the most frequently used transformations in the 100 winning sequences. These were loop peeling, partial redundancy elimination, logical peephole optimization, register-to-register copy coalescing, and useless control-flow elimination. Drawing on the one-letter codes for these transformations (see Table 1), we call this set `plosn`. The other space that we use in these experiments is `pdxnt`, which was hand-picked to be stronger than `plosn`.

To date, we have enumerated 6 subspaces using 4 programs and 2 sets of optimizations: `fmin+plosn`, `zeroin+plosn`, `adpcm-d+plosn`, `svd+plosn`, `fmin+pdxnt`, and `zeroin+pdxnt`. The first enumeration required 14 CPU months to complete. With engineering improvements, we have reduced that time to roughly 2.5 CPU months or two calendar weeks on a small collection of servers. The resulting data, representing nearly 60,000,000 combinations of sequence-plus-application, has been analyzed for properties that are useful for a search algorithm. The following properties hold across all the enumerated spaces.

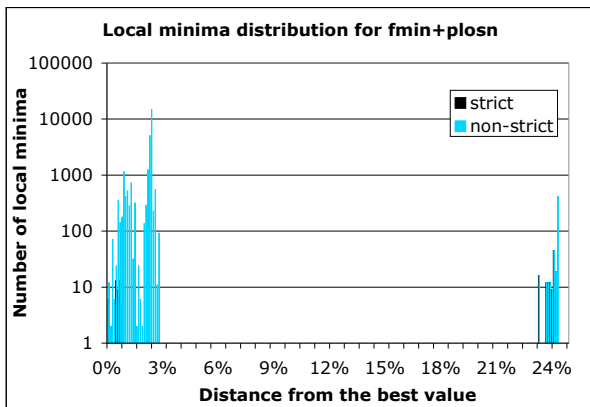
Range of Improvement In our subspace enumerations, we routinely see variations of 70% or more between the best sequence and the worst. Figures 2a and b show the dis-



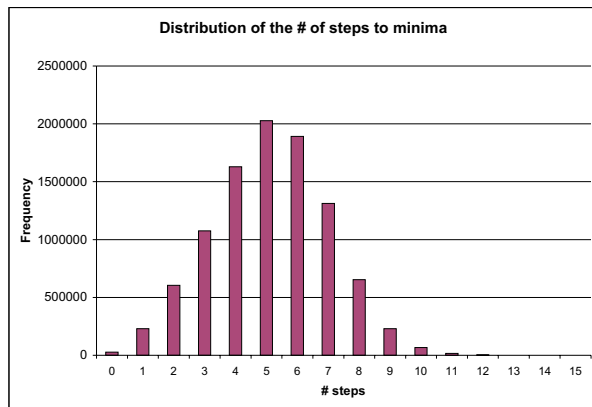
(a) Distribution of values in plosn



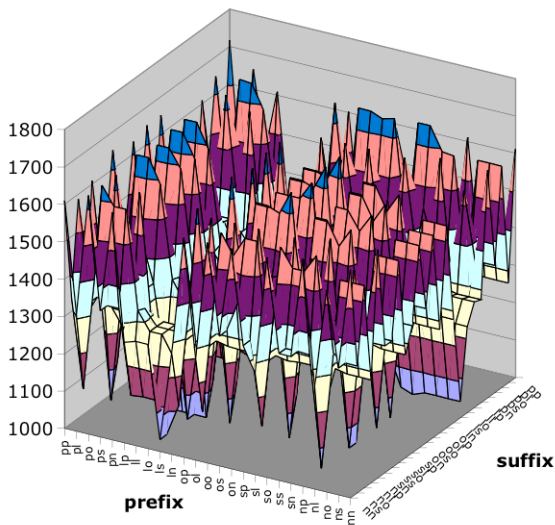
(b) Distribution of values in pdxnt



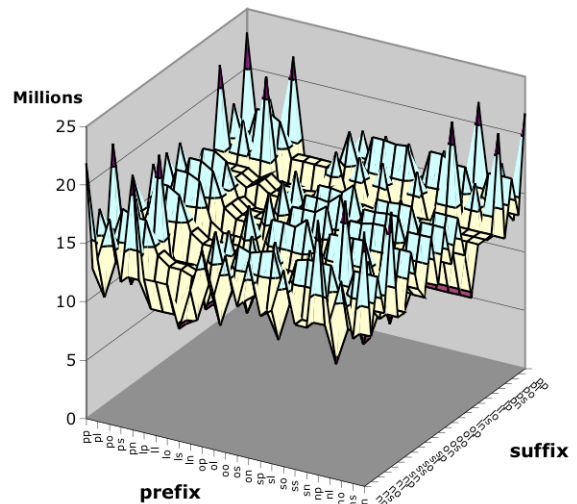
(c) Distribution of local minima in fmin+plosn



(d) Downhill steps to a minimum in fmin+plosn



(e) 4-of-5 surface for fmin+plosn



(f) 4-of-5 surface for adpcm-c+plosn

Figure 2: Observed Properties of the Enumerated Spaces

- c Sparse conditional constant propagation [26]
- d Dead code elimination based on SSA-form [10, 8]
- g Optimistic value numbering [2]
- l Partial redundancy elimination [20]
- m Renaming builds the name space needed by the implementations of `l` and `z`. The compiler inserts it automatically before `l` or `z`.
- n Useless control-flow elimination [8]
- o Peephole optimization of logically adjacent operations [11]
- p Peel the first iteration of each innermost loop
- r Algebraic reassociation [3]
- s Register-to-register copy coalescing [4]
- t Operator strength reduction [8]
- u Local value numbering [8]
- v Optimistic global value numbering [22]
- x Dominator tree value numbering [8]
- y Extended basic-block value numbering [8]
- z Lazy code motion [17]

Table 1: Transformations in the 10-of-16 space

tribution of values for two programs, `fmin` and `zeroin` in the subspaces `plosn` and `pdxnt`. To calibrate the data, the best sequence in `fmin+plosn` is about 18% slower than the best sequence that we have found in the 10-of-16 space for `fmin`. The best sequence in `fmin+pdxnt` is about 20% slower than the best sequence in `fmin+plosn`. Recall that we hand-picked `pdxnt` and expected it to be superior to `plosn`, showing again that optimization choice is difficult.

In some subspaces, the worst sequence produces the same code as running no optimization passes; in others, the worst sequences produce slower code than no optimization. With our compiler, we often see code that executes 15 to 25% fewer operations than the base sequence for our compiler. Other authors report similar improvements [13, 24].

Local Minima The spaces have many local minima, distributed across a wide range of values. Figure 2c shows the distribution of local minima in `fmin+plosn`. This subspace has 258 strict local minima, points where every neighboring point has a higher value, and 27,315 non-strict local minima, points where each neighbor has a greater or equal value.

Distance to Local Minimum The distance to a local minimum from a randomly-chosen point is small. Figure 2d shows the distribution of the number of downhill steps needed to reach a local minimum, taken over all 9,765,625 points in `fmin+plosn`. Notice that most downhill walks reach a minimum within eight steps and all of them halt within fifteen steps, which suggests that hillclimbers should halt quickly.

Surfaces in the Search Space The search spaces in which the prototype operates are neither smooth nor convex. While intuition and experience suggest this result, the enumeration studies demonstrate it. To show this graphically, Figures 2e and f present 4-of-5 spaces for `fmin+plosn` and `adpcm+c+plosn`, respectively. The lower axes show two character prefixes and suffixes. The function values show the number of operations executed by the resulting code.

Clearly, the order assigned to the five transformations affects the picture that is generated for the 4-of-5 subspaces. No consistent ordering of the two lower axes produces a smooth picture; the location of points shifts, but the topography does not.

Section Summary From our enumeration experiments, we have learned that these spaces have many local minima. The minima are distributed across a range of function values. The distance to a local minimum from a random point is small. The surfaces have sharp features that are hostile to simple search techniques, such as a line search. Using this knowledge, we expect to improve on the effectiveness our searches. Because of the demonstrated success of GAs on sequence-finding problems [7, 19], we will compare our new methods against our best GA.

4. SEARCH ALGORITHMS

The primary costs involved in finding compilation sequences arise from compiling the code and evaluating the quality of the resulting code. Both these costs rise directly with the number of sequences that the compiler evaluates. Improving the efficiency of the compiler’s search algorithms directly reduces the cost of sequence finding. Thus, we have focused a large part of our effort on improvements in sequence finding. This section describes three different techniques for finding good sequences: an improved genetic algorithm, a greedy constructive algorithm, and a family of hillclimbers.

4.1 Improved Genetic Algorithm

Genetic algorithms often work well in complex discrete spaces (recall Figure 2e and f). Schielke’s original work used a GA with a population of 20 strings, a single-point random crossover, survival of the single best string at each generation, and a low mutation rate [7]. Since that original paper, we have experimented with many variations in the GA’s parameters and behaviors. Our current “best” GA has the following characteristics:

Population: Each generation consists of 50 strings.

Elitism: The best 10% always survive intact.

Reproduction: The remaining 90% of the next generation is created with the crossover mechanism.

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

Crossover: It uses a single-point random crossover.

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

Duplication: Hashing detects previously seen strings. Duplicates are mutated into untested strings.

We show results for this GA running for 50 evolutionary steps (GA-50×50) and for 100 steps (GA-50×100).

GAs are the method of choice in discrete combinatorial spaces where the distribution and density of good solutions is unknown. They are particularly cost effective for problems where the solution density is very low. The nonlocal nature of the crossover operation lets a GA explore far flung regions in the solution space.³

4.2 Greedy Constructive Method

Greedy methods often work well for complex problems. To assess the prospect for greedy techniques in the sequence-selection problem, we built a greedy sequence constructor.

³We expect this power to be more important when sequence selection is applied to high-level optimizations (*e.g.*, loop blocking or unroll-and-jam) than it is when applied to the low-level scalar optimizations used to date in our work.

```

      ps      pps
      ls      lps
p     os      ops
l     ss      sps
o  => ns  => nps  => ...
s  s  sp  ps  psp  ops
n     sl      psl
      so      pso
      sn      pss
           psn

```

Figure 3: A Greedy Constructive Run

The greedy constructive method (GC) works on a simple paradigm, extending a good sequence one character at a time. Figure 3 shows how the algorithm might run in a `plosn` subspace.

The initial step tries each of `p`, `l`, `o`, `s`, and `n`. It selects the pass that most improves the code, say `s`. Next, it tries each pass as a prefix to the current string and as a suffix to the current string. In `plosn`, this leads to nine two-character strings. (`ss` occurs twice.) It chooses the best of these strings, say `ps`, and repeats the process to derive a three-character string. This process continues until either a string of the desired length is found or no extension produces an improvement.

The GC algorithm, as stated, has bounded cost. For a pool of k transformations, the initial step performs k evaluations. Each subsequent step performs at most $2k$ evaluations. For a string of length m , it should perform $2mk - k$ evaluations. In the 10-of-16 space, that implies at most 303 evaluations.⁴ At this cost, GC represents a significant reduction from the 10,000 to 20,000 evaluations that Schielke used.

Unfortunately, equal-valued strings (ties) complicate GC’s behavior. If GC exhaustively pursues all ties (GC-EXH), the number of evaluations grows wildly. If GC pursues ties in a breadth-first fashion (GC-BF), the number of evaluations is much smaller than with GC-EXH. An alternate strategy has GC select one of the tied values at random and continue. Random tie-breaking produces strings more efficiently; to compensate for its more narrow search, we can run it multiple times (GC-10 and GC-50).

adpcm-d	GC-EXH	GC-BF	GC-10	GC-50
Trials	936,222	423	1,567	3,573
Speed	70.01%	71.64%	70.01%	70.01%

This table shows the impact of the three GC tie-breaking strategies for the program `adpcm-d` from MediaBench. The `Trials` line shows the number of sequences evaluated and the `Speed` line indicates the number of operations executed as a percentage of the operations executed by code compiled with the compiler’s default sequence (see Section 5). Because `adpcm-d` produces myriad ties, the cost of GC-EXH explodes. GC-BF examines the fewest sequences, but sacrifices a minor amount of code quality. Both GC-10 and GC-50 evaluate more sequences than GC-BF. In this case, they both produce the same code as GC-EXH, with much less effort.

⁴The two character strings always have one duplicate. Occasional duplicates are possible, but unlikely, at longer strings.

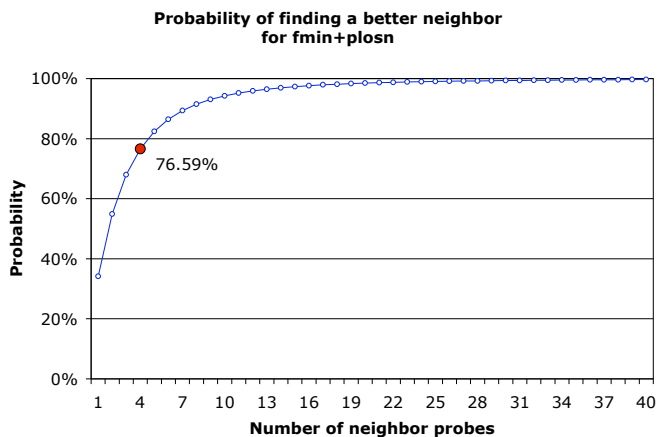


Figure 4: Probability of Downhill Step vs. Patience

4.3 Hillclimbers

A hillclimber starts at some initial point, evaluates its neighbors, and moves to a point with a better fitness value. It continues in this manner until it gets stuck in a (local) minimum. The notion of “neighbor” is fundamental to a hillclimber. We define the neighbors of a string x as all strings of the same length that differ in one position from x —that is, strings at Hamming distance one from x .

The craggy nature of our spaces, shown in Figure 2e and the short distance to a local minimum, shown in Figure 2d, both suggest that hillclimbers may be effective.⁵ We have experimented with several kinds of hillclimbers in these spaces.

Steepest Descent: At each step, the hillclimber evaluates each neighbor’s fitness value. It moves to the neighbor with the best value. If no neighbor has a better value than the current point, it halts.

Random Descent: At each step, the hillclimber evaluates neighbors in random order. If it finds a better fitness value, it moves to that neighbor without looking further. Otherwise, it halts.

Impatient Random Descent: The hillclimber follows random descent with one added rule. It evaluates at most *patience*% of the neighbors before deciding that the current point is a local minimum.

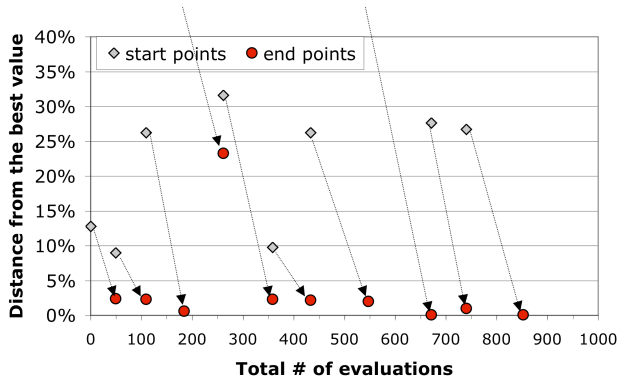
These three variants differ in the number of neighbors that they evaluate. In 10-of-5 space, a point has 40 neighbors, while in 10-of-16 space, it has 150 neighbors. Steepest descent evaluates each neighbor before taking a step. Random descent can evaluate many fewer points; in the worst case (and for the final point), it examines each of a point’s neighbors. The impatient random algorithm explicitly bounds on the work that it expends before declaring that the current point is a (local) minimum.

Understanding Impatience In practice, impatient random hillclimbers perform well for the sequence selection problem. Figure 4 suggests the reason. It plots the probability of finding a downhill step against the number of neighbors examined, in `fmin+plosn`. At a patience level of 10% (4 neighbors

⁵The measured distance to a local minimum assumes Hamming-1 neighbors.

in 10-of-5 space) the probability of finding a downhill step is $> 76\%$. That is, impatient descent stops prematurely less than 24% of the time with a 10% patience level. Larger patience levels decrease the likelihood of a premature halt and increase the cost of each step. (The probabilities shown are for each individual descent step.)

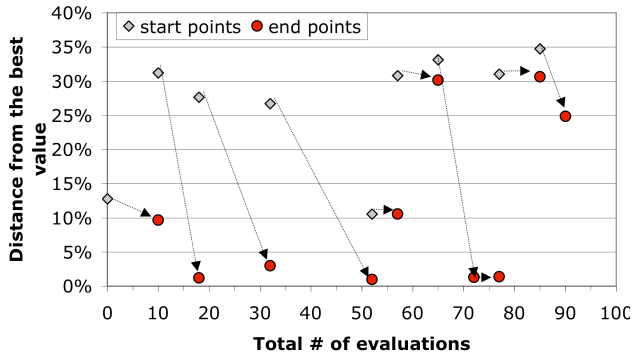
Impatience, then, is a tradeoff between the likelihood of a premature halt and the number of neighbors that the algorithm examines. In `fmin+plosn`, a 10% impatient, random descent run should terminate at a good local minimum 40% of the time [6]. To see the effect of impatience, consider the following two plots. The first shows the progress of 10 runs of a patient random descent algorithm in `fmin+plosn`.



10 Runs of Patient Random Descent in `fmin+plosn`

The patient algorithm finds good solutions on most of its 10 runs. One run finishes with a bad local minimum; the rest finish within 4% of the optimum and several finish within 1% of optimum.

The second plot shows 10 runs of an impatient random descent algorithm, with 10% patience, in `fmin+plosn`. (We call this method HC-10).



10 Runs of 10% Impatient Random Descent in `fmin+plosn`

With impatient descent, half of the runs finish far from the optimum – more than 10% larger than the optimum. The other half, however, find sequences that are close to the optimum. In comparing the two plots, notice the total number of evaluations required. The patient algorithm requires more than 800 evaluations, while the impatient algorithm uses fewer than 100.⁶

⁶These runs are performed in a 10-of-5 space, where each

Name	Suite	# of Proc's	# of SLOC	# of Blocks
<code>fmin</code>	FMM	1	160	59
<code>zeroin</code>	FMM	1	125	39
<code>svd</code>	FMM	1	351	185
<code>adpcm-c</code>	Media	1	192	37
<code>adpcm-d</code>	Media	1	168	30
<code>g721-c</code>	Media	16	965	268
<code>g721-d</code>	Media	21	1080	344
<code>fpppp</code>	SPEC	11	2620	380
<code>tomcatv</code>	SPEC	1	192	78
<code>nsieve</code>	—	2	117	38

Table 2: Benchmark programs

Increasing the patience level to 20% decreases the likelihood that a given descent run halts prematurely but potentially increases the cost of that run. At each point, a 20% patient run may evaluate twice as many neighbors. The increased probability of finding a downhill step also means that it may take more steps. In `fmin+plosn` and `zeroin+plosn`, a 10% patience level produces an average of 1.9 descent steps per run. At 20% patience, the runs increase to 4 steps. In the worst case, 20% patience run evaluates four times as many sequences as a 10% run – in the 10-of-5 spaces. This effect is large in the 10-of-16 spaces, where each string has 150 neighbors. There, 10% patience can examine 15 neighbors while 20% patience can examine 30. This argument suggests using smaller patience factors and more restarts; experimentation confirms the merit of this approach.

HC-10, an impatient (10%) random descent, represents a compromise between random probing of the space and an impatient search for a local minimum. Impatience capitalizes on the high probability of finding a better neighbor. Random restart capitalizes on the relative abundance of good solutions.

Hillclimbers Versus Random Probing Hillclimbers proceed by making successive local improvements to a solution. They are effective in search spaces with a low to moderate solution density. High solution density makes random probing a viable alternative; with enough good solutions, random probing becomes the method of choice. Our experiments to date indicate that random probing is less effective than guided searches such as hillclimbers for the benchmarks and optimizations studied in this paper [1, 6]. The experimental data (see Tables 3 and 4) shows that impatient hillclimbing, with its combination of random restart to get near a good solution and local search to find local improvements, is sufficient to find good local minima.

5. EXPERIMENTAL RESULTS

We evaluated the performance of the greedy constructive algorithm, the impatient random hillclimber, and the genetic algorithm on a small suite of benchmarks, described in Table 2. First, we compared derived sequence quality for the three approaches, noting the effort that each expended. Next, we studied their evaluation-efficiency by examining sequence quality achieved with a fixed number of trials.

point has 40 neighbors. Thus, the number of trials is much smaller than the impatient random descent runs for `fmin` shown in Section 5. The experiments in Section 5 are conducted in a 10-of-16 space, with 150 neighbors per point.

10-of-16 Space	GC-10		GC-BF		HC-10		HC-50		GA-50	
	Ops	Trials	Ops	Trials	Ops	Trials	Ops	Trials	×50	×100
<i>fmin</i>	88.3%	588	88.3%	396	74.0%	413	73.9%	2,124	73.5%	73.5%
<i>zeroin</i>	70.4%	1,377	71.2%	814	74.0%	462	71.0%	2,054	69.6%	69.6%
<i>adpcm-c</i>	68.0%	838	69.7%	524	70.4%	402	68.5%	2,238	67.0%	67.0%
<i>adpcm-d</i>	70.0%	1,567	71.6%	423	70.0%	423	70.0%	2,328	69.4%	69.4%
<i>g721-e</i>	84.1%	303	84.1%	303	85.3%	502	83.3%	2,857	81.3%	81.3%
<i>g721-d</i>	83.9%	303	83.9%	303	82.2%	427	82.1%	2,752	81.5%	80.8%
<i>fp PPP</i>	81.1%	678	81.1%	458	78.8%	632	75.3%	3,220	75.9%	75.3%
<i>nsieve</i>	57.5%	1,859	57.5%	566	57.5%	391	57.5%	2,452	57.5%	57.5%
<i>tomcatv</i>	120.3%	303	120.3%	303	85.4%	458	83.0%	2,555	85.1%	82.7%
<i>svd</i>	77.6%	545	77.6%	755	77.1%	631	75.8%	2,756	74.0%	73.9%
<i>geo mean</i>	79.6	—	79.1	—	75.0	—	73.6	—	73.1	71.6

Table 3: Runs to Completion in 10-of-16 Space, Relative to the Fixed Sequence `rvzcodtvtzcod`

Benchmark Programs Table 2 shows the benchmark programs discussed in this section. The final three columns list the number of procedures, number of source-code lines, and number of basic blocks for each program. *FMM* is the library of code distributed with Forsythe et al. [12]. *Media* refers to the MediaBench suite. *SPEC* denotes the SPEC 95 benchmarks. *nsieve* is the classic sieve of Eratosthenes benchmark written by Al Aburto.

We began our experiments with *fmin*, a small program that runs quickly. As we have improved the search algorithms and decreased the costs of compilation, we have increased the size and running time of the test codes.

Execution Environment All tests reported in this paper measure performance in terms of operations executed on a simulated RISC machine. We use this metric for several reasons:

1. These results have been gathered over a four-year period, using a variety of machines. Simulated operations is a metric that is stable in the long term and can be obtained using almost any available hardware.
2. Simulated operations is a precise metric – it does not vary between runs. Measured execution time shows a variance, on several of the systems that we use, of 1 to 3%. That measurement error is larger than some of the differences that the search algorithms encounter.
3. We have run experiments using other metrics, including code size, interoperation bit-transitions, and running time of native SPARC code. Those experiments confirm the conclusions about search algorithms reached through the experiments reported here.

The purpose of these experiments is to learn about search algorithms. The number of operations executed relates to execution time, particularly in regard to the low-level, scalar optimizations used in these experiments. It produces consistent measurements. It induces search spaces with the same properties that we would see with consistent running times.

Training Bias We have performed training-testing studies on the results obtained by the search algorithms. In these experiments, we see no systematic bias in the results – the testing data achieves results within ± 1 to 2% of the training data.

Finding Good Sequences Table 3 shows data on six search algorithms, run on the ten programs described earlier. The search algorithms are:

GC-10 & GC-BF are greedy constructive searches. The former uses depth-first search and random tie-breaking with 10 trials. The latter uses breadth-first search.

HC-10 & HC-50 are random, impatient hillclimbers with 10% patience. The former performs 10 trials, while the latter does 50 trials.

GA-50×50 & GA-50×100 run the genetic algorithm on a pool of 50 strings. **GA-50×50** runs for 50 evolutionary steps. **GA-50×100** runs for 100 steps

Table 3 shows the results of running six search algorithms to their completion in the 10-of-16 space. The **Ops** column shows the number of operations executed, expressed as a percentage of the number of operations executed by code prepared with the compiler’s default sequence, `rvzcodtvtzcod`. The default sequence, which uses 12 passes, produces a significant reduction in operations executed from the unoptimized code. The **Trials** column shows the number of sequences that the search algorithm evaluated to find the solution shown in the **Ops** column. **Trials** numbers are omitted for the GAs because those numbers are constant: 2,300 for **GA-50×50** and 4,550 for **GC-50×100**. The bottom row shows the geometric mean of each **Ops** column.

The table divides the methods into the inexpensive methods, **GC-10**, **GC-BF**, and **HC-10**, and the expensive methods, **HC-50** and the two GAs. For the inexpensive methods, we have highlighted the best result. Both **GC-10** and **HC-10** achieve some good results. Looking at the geometric mean for the improvements, **HC-10** is the clear winner among the inexpensive methods. Both versions of **GC** are hurt by the poor performance on *tomcatv*. Its costs are fairly consistent, between 402 and 632 trials. The effort required by **GC-BF** is consistent within a broader range, and **GC-10** shows a broad variation in effort. (See Section 6.)

The data for the more expensive methods shows that additional improvement is possible. Both **HC-50** and **GA-50×50** expend about the same effort. The geometric mean suggests that **GA-50×50** makes slightly better use of those cycles. **GA-50×100** finds a 1.5% improvement with roughly double the effort.

Efficiency and Effectiveness To refine our understanding of the tradeoff between cost and solution quality, we ran 1,000 trials each of **GC** with random tie-breaking, **HC** with 10% impatience, and **GA-50**. Table 4 records the solution quality achieved by each approach after 100, 250, 500, and 1,000 trials. As in Table 3, solution quality is expressed

10-of-16 Space	100 Trials			250 Trials			500 Trials			1,000 Trials		
	GC	HC	GA	GC	HC	GA	GC	HC	GA	GC	HC	GA
fmin	89.49	84.83	87.41	88.35	83.13	80.55	88.35	77.05	77.20	88.29	76.58	75.41
zeroin	85.48	79.31	83.03	75.94	78.94	76.82	71.54	77.40	75.93	71.54	72.26	73.89
adpcm-c	70.29	81.10	75.42	70.09	79.22	74.13	69.50	72.43	72.07	68.16	69.32	69.61
adpcm-d	70.58	90.79	74.13	70.57	80.57	73.68	70.02	70.04	71.65	70.01	70.04	71.22
fp PPP	89.10	85.10	92.82	83.21	81.60	90.50	82.53	80.34	80.91	81.13	78.38	78.13
nsieve	57.47	73.58	64.98	57.47	57.47	64.86	57.47	57.47	57.47	57.47	57.47	57.47
tomcatv	125.26	101.73	104.73	120.40	92.61	99.00	120.29	88.16	93.04	<i>120.29</i>	87.33	89.28
svd	96.86	93.44	95.64	94.92	87.12	88.54	77.64	82.45	80.59	77.64	78.26	77.47
geo mean	83.48	85.84	83.88	82.62	80.09	81.01	77.95	75.15	75.5	75.58	73.23	73.57

Table 4: Progress in the 10-of-16 Space, Relative to the Fixed Sequence rvzcodtvzcod

as a percentage of the operations executed by the code as compiled with the fixed sequence. At each level of effort, for each benchmark, the best result appears in boldface.

For `tomcatv`, the GC algorithm encounters no ties. Thus, it explores the space completely in 303 trials. This result is listed in the 500 trials column and in the 1,000 trials column. The 1,000 trial entry is printed in pale italics to remind the reader of its unusual nature.

If we simply consider which algorithm wins most often, the data is mixed. The geometric means, however, show a more nuanced view. At 100 trials, GC beats HC and edges out GA. At 250, 500, and 1,000 trials, HC beats the other algorithms. The GA narrows the gap between it and HC from 500 to 1,000 trials. As we know from Table 3, by 2,300 trials, GA-50 is the method of choice.

Section Summary These results suggest the use of different techniques at different levels of effort. At a mere 100 trials, the GC algorithm performs pretty well. In the neighborhood of 400 to 600 trials, HC-10 (run to completion) does well and achieves results that are within a couple percent of the expensive methods. For situations where performance is truly critical, the extra cost of using GA-50×50 may be justified.

The progressive nature of both HC and GA-50 may make them more useful, in practice, than GC. In the range of 250 to 1,000 trials, HC does well; the compiler can let it run for fixed amount of time or a fixed number of trials. Somewhere between 1,000 and 2,000, the balance shifts to GA-50. It will continue to find improvements out to at least 4,500 trials. These two algorithms can form the foundation for a practical time-limited or effort-limited search.

6. COST-IMPROVEMENT ANALYSIS

Figure 5 plots improvement versus evaluations for GC-10, HC-10, HC-50, and GA-50×50. The runs of HC-10 form a cluster, centered around 500 trials and showing improvement between 15 and 30%. From a cost-benefit perspective, this region in the chart is best. The runs for HC-50 and GA-50×50 fall into another cluster to the right of HC-10, between 2,000 and 3,000 trials. These points show small improvements in performance over HC-10, achieved at markedly higher cost. The results for GC-10 do not cluster. Instead, they spread across the graph, from 300 to 1,600 trials.

This figure clearly shows both the consistent, cost-effective behavior of HC-10, and the variability of GC-10. It also dramatically illustrates the jump in cost between the HC-10 cluster and the cluster for HC-50 and GA-50×50.

7. PRIOR WORK

Several groups have worked on problems related to finding compilation sequences. Whitfield and Soffa characterized and modeled the interactions among optimizations [27]. Schielke’s 1999 paper appears to be the first use of a GA to find compilation sequences; he showed improvements in both code size and execution speed [7].

In VISTA Kulkarni *et al.* used a GA, performance information, and user input to select a sequence of optimizations, both local [19] and global [18]. Their 2004 paper describes techniques to recognize when evaluation is not necessary [18]; those techniques should be directly applicable to both HC and GA-50. Similarly, HC-10 should work in their system as a substitute for the GA.

Triantifyllis *et al.* demonstrate the promise of using multiple sequences in a practical system; they built their *optimization space exploration* (OSE) algorithm into Intel’s Electron compiler for the Itanium. OSE is only applied to hot code sequences. Their version of Electron tries up to 12 compilation sequences and keeps the best result [24].

Zhao *et al.* describe an approach for modeling interactions in a predictive framework [28]; subsequent work (in press) shows successful application of their framework to predict profitability of individual optimizations.

Kisuki *et al.* used parameter sweeps, genetic algorithms, and direct search to find good optimization settings for loops [16, 15, 13]. Their work focused on numerical kernels.

Feedback-based adaptation has been applied within single transformations, as well. Motwani *et al.* used alpha-beta tuning in an algorithm that combined instruction schedul-

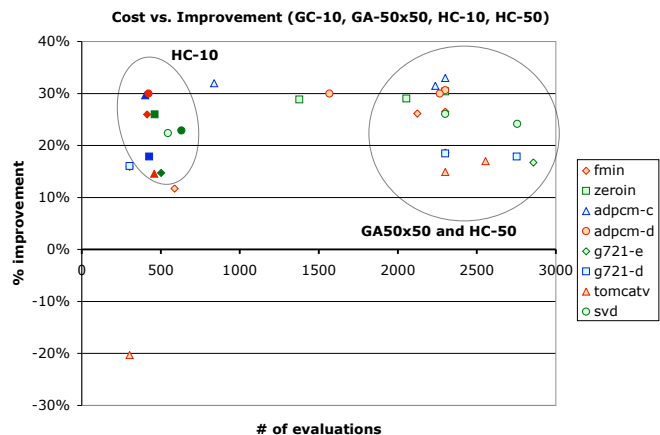


Figure 5: Cost-Improvement Tradeoffs

ing and register allocation [21]. Stephenson *et al.* used genetic programming to derive improved heuristics for hyperblock formation, for register allocation and for data prefetching [23]. Working in the MIPSPro compiler, Waterman showed that simple feedback-based search techniques can find better loop-blocking sizes than the compiler's default heuristic [9].

Several authors have looked at adaptive selection of command-line parameters. Granston and Holler developed an algorithm to pick program-specific or file-specific command-line options for the HP PA-RISC compiler [14]. Chow and Wu used a fractional factorial design to attack the same problem in an Intel research compiler for the IA-64 [5].

Conclusions

The algorithms described in this paper are able to find good compilation sequences with roughly 400 to 600 probes of the search space – an order of magnitude reduction from prior work and a factor of four reduction from our best results with genetic algorithms. These search techniques, particularly the impatient random descent algorithm with ten restarts, have reached a level of efficiency at which adaptive, feedback-based selection of optimization sequences becomes practical for performance-critical applications.

Two of the three algorithms describes in this paper are progressive. Given more time, they produce better results. Stopped at an arbitrary time, they produce the best result seen to that point. The HC algorithm works well over the range of effort from 400 probes to 1,000 probes. By 2,000 probes, the additional power of GA-50 begins to show. We believe that HC-10 is a good algorithm for routine use; it achieves significant improvements over fixed-sequence compilation. However, for the final compilation of a performance-critical application, running GA-50 for a minimum of 2,300 evaluations may produce better results. Adding effort beyond that point may produce further marginal improvements.

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.

8. REFERENCES

- [1] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Proceedings of ACM SIGPLAN Conference on Languages Tools and Compilers for Embedded Systems*, pages 231–239, June 2004.
- [2] 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.
- [3] 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*.
- [4] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via graph coloring. *Computer Languages*, 6(1):47–57, Jan. 1981.
- [5] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In *Proceedings of the 4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, Dec. 2001.
- [6] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Exploring the structure of the space of compilation sequences using randomized search algorithms. In *Proceedings of the 2004 Symposium of the Los Alamos Computer Science Institute*, Oct. 2004. (Proceedings distributed electronically to attendees. Copy available at <http://www.cs.rice.edu/~keith/Adapt.>)
- [7] 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.
- [8] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan-Kaufmann Publishers, 2003.
- [9] K. D. Cooper and T. Waterman. Investigating adaptive compilation using the MIPSPro compiler. In *Proceedings of the 2003 Los Alamos Computer Science Institute Symposium*. Los Alamos Computer Science Institute (LACSI), Oct. 2003.
- [10] 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.
- [11] J. W. Davidson and C. W. Fraser. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(2):191–202, Apr. 1980.
- [12] G. E. Forsythe, M. A. Malcolm, and C. B. Moler. *Computer Methods for Mathematical Computations*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1977.
- [13] G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In *Proceedings of the Fifteenth Workshop on Languages and Compilers for Parallel Computers (LCPC 02)*, July 2002.
- [14] E. D. Granston and A. Holler. Automatic recommendation of compiler options. In *Proceedings of the 4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, Dec. 2001.
- [15] T. Kisuki, P. M. Knijnenburg, and M. F. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT'00)*, pages 237–248, Oct. 2000.
- [16] T. Kisuki, P. M. Knijnenburg, M. F. O'Boyle, and H. Wijsho. Iterative compilation in program optimization. In *Proceedings of 8th Workshop on Compilers for Parallel Computers, CPC 2000*, pages 35–44, Jan. 2000.

- [17] J. Knoop, O. Rütting, 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*.
- [18] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Searches for effective optimization phase sequences. *ACM SIGPLAN Notices*, 39(6):171–182, May 2004. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*.
- [19] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Park, and K. Gallivan. Finding effective optimization phase sequences. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages Compilers, and Tools for Embedded Systems (LCTES 03)*, pages 12–23, June 2003.
- [20] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, Feb. 1979.
- [21] R. Motwani, K. V. Palem, V. Sarkar, and S. Reyen. Combining register allocation and instruction scheduling. Technical Report 698, Courant Institute of Mathematical Sciences, New York University, July 1995.
- [22] L. T. Simpson. *Value-Driven Redundancy Elimination*. PhD thesis, Rice University, 1996.
- [23] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. *SIGPLAN Notices*, 38(5):77–90, May 2003. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
- [24] S. Triantifyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *Proceedings of the First International Symposium on Code Generation and Optimization*, Mar. 2003.
- [25] T. Waterman. Adaptive compilation and inlining. Thesis proposal, Rice University, Oct. 2004.
- [26] M. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr. 1991.
- [27] 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.
- [28] M. Zhao, B. Childers, and M. Soffa. Predicting the impact of optimizations for embedded systems. In *Proceedings of ACM SIGPLAN Conference on Languages Tools and Compilers for Embedded Systems*, pages 1–11, June 2003.
- [29] W. Zhao, B. Cai, D. Whalley, M. Bailey, R. Van Engelen, X. Yuan, J. D. Hiser, J. W. Davidson, K. Gallivan, and D. L. Jones. VISTA: A system for interactive code improvement. In *Proceedings of the ACM SIGPLAN Joint Conference Languages, Compilers and Tools for Embedded Systems (LCTES 02) and Software and Compilers for Embedded Systems (SCOPES 02)*, pages 155–164, June 2002.