Finding Effective Compilation Sequences

L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey
Steven W. Reeves, Devika Subramanian, Linda Torczon, Todd Waterman

Rice University
Houston, Texas, USA

ABSTRACT

Most modern compilers operate by applying a fixed, program-independent sequence of optimizations to all programs. Compiler writers choose a single "compilation sequence", or perhaps a couple of compilation sequences. In choosing a sequence, they may consider performance of benchmarks or other important codes. These sequences are intended as general-purpose tools, accessible through command-line flags such as `-O2` and `-O3`.

Specific compilation sequences make a significant difference in the quality of the generated code, whether compiling for speed, for space, or for other metrics. A single universal compilation sequence does not produce the best results over all programs [8, 10, 29, 32]. Finding an optimal program-specific compilation sequence is difficult because the space of potential sequences is huge and the interactions between optimizations are poorly understood. Moreover, there is no systematic exploration of the costs and benefits of searching for good (i.e., within a certain percentage of optimal) program-specific compilation sequences.

In this paper, we perform a large experimental study of the space of compilation sequences over a set of known benchmarks, using our prototype adaptive compiler. Our goal is to characterize these spaces and to determine if it is cost-effective to construct custom compilation sequences. We report on five exhaustive enumerations which demonstrate that 80% of the local minima in the space are within 5 to 10% of the optimal solution. We describe three algorithms tailored to search such spaces and report on experiments that use these algorithms to find good compilation sequences. These experiments suggest that properties observed in the enumerations hold for larger search spaces and larger programs. Our findings indicate that for the cost of 200 to 4,550 compilations, we can find custom sequences that are 15 to 25% better than the human-designed fixed-sequence originally used in our compiler.

This work has been supported by NSF ITR Grant CCR-0205303 and by the Los Alamos Computer Science Institute.

LCTES'04, June 11–13, 2004, Washington, DC, USA.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers and Optimization*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search

General Terms

Experimentation, Languages, Performance

Keywords

Adaptive Compilers, Learning Models

1. INTRODUCTION

The basic structure of a compiler has been frozen since the late 1950s. It consists of a fixed set of passes that run in one of a handful of preselected sequences (e.g., `-g`, `-O1`, `-O2`, ...). After forty or more years of research, we have at our disposal a huge collection of optimizations, but the fixed structure of our compilers cannot use these techniques to their greatest effect. In LCTES 99, we showed that choosing a set of optimizations and an order for them in a program-specific way could produce better code, whether optimizing for runtime speed or for code space [8]. Embedded systems are a natural application area for such compilation techniques, because the developers are often willing to pay additional costs to meet specific performance goals. Since that paper, several authors have looked at related problems or have built systems that work along similar lines [20, 21, 29, 23, 33].

Our work in LCTES 99 used a particularly simple genetic algorithm (GA) to choose compilation sequences. To improve on that method, we need to understand the huge discrete search spaces in which a compiler that tries to find custom compilation orders operates. The compiler used in this paper has 16 distinct transformations, ignoring parameter settings. Choosing sequences of length 10 produces a space containing 16^{10}, or 1,099,511,627,776, sequences. (Letting the search vary the parameters to the optimizations would create an even larger space.) If the compiler is finding program-specific sequences, each program will generate its own search space. Before we can build practical compilers that operate in such spaces, we need a fundamental understanding of these search spaces and their properties.

This paper presents the results of a large experimental study that characterizes the search spaces in which such a compiler operates. It describes our prototype adaptive compiler. It reports on large-scale experiments, including complete enumerations of some limited subspaces and sparse samplings of larger spaces. It discusses the efficacy of several

search algorithms that we have designed to capitalize on the measured properties of these spaces.

To our knowledge, this paper is the first experimental characterization of these search spaces. Some of the results are intuitive; for example, these spaces are neither smooth nor continuous, so that simple steepest descent methods are not guaranteed to reach a global minimizer. Other results are surprising; for example the distance from a randomly chosen point to a local minimum is small (see Figure 6).

Section 2 discusses prior work on the problem of finding custom compilation sequences. Section 3 describes the enumeration experiments that we have performed to understand the properties of the search spaces. Section 4 presents several search strategies and evaluates their efficacy on a set of benchmark programs with our prototype compiler. The work in this section uses much larger search spaces than the enumerations in Section 3. The final section discusses some of the open problems that must be solved to make adaptive compilers practical.

2. PRIOR WORK

A growing body of literature suggests that adaptive behavior can improve program performance. This work falls, roughly, into three areas: programmed adaptation in libraries, algorithms to find good compilation sequences, and attempts to adaptively control the compilation process using command-line parameters.

Several groups have produced adaptively self-tuning numerical libraries. This work suggests the kind of improvement that might be achieved with adaptive compilation. The ATLAS library chooses, at runtime, the most appropriate implementation from a set of precompiled versions; the installation runs a series of machine-specific performance tests that compute values for runtime parameters such as blocking sizes [31]. Both FFTW and UHFFT generate custom-optimized codes for performing fast Fourier transforms [17, 16, 25]. The numerical libraries for the Thinking Machines CM-2 and CM-5 machines use metrics based on problem size to make algorithm-choice decisions [19].

Other authors have looked at choosing compilation sequences. Kulkarni *et al.* used a GA, performance information, and user input to select a sequence of local optimizations in VISTA [23]; their GA is modeled after Schielke's GA [8]. Their system might well be improved by using the search techniques described in Section 4. Zhao *et al.* are developing analytical models that can be used to steer the selection of a compilation sequence [33]. In the long run, models like those that they propose can replace parts of the feedback cycle that our system uses. Triantifyllis *et al.* recognize the potential benefits of finding good sequences; to limit compile time, their system uses a fixed set of sequences and retains the best results [29].[1] The explicit goal of our work on search algorithms is to achieve similar efficiency without constraining the compiler to pre-selected sequences.

Several groups have looked at the problem of adapting compiler behavior by picking program-specific command-line parameters. Granston and Holler developed an algorithm that picked program-specific or file-specific command-line options for the HP PA-RISC compiler [18]. Chow and Wu used a fractional factorial design to attack the problem in Intel's compiler for the IA-64 [7]. Knijnenburg *et al.* adopted a different approach; they used a parameter sweep to find the best blocking factors for loops [21, 20]; we have shown similar results with search-based techniques that use fewer probes of the search space [12].

[1]This approach resembles the *best-of-three* spilling heuristic proposed for register allocation by Bernstein *et al.* [3].

3. CHARACTERIZING SEARCH SPACES

Prior work has demonstrated that automatically-chosen, program-specific compilation sequences can produce significantly different results for a variety of objective functions, including both the runtime speed and size of the compiled code [8, 10]. The techniques used to find such sequences, however, are too expensive to make such adaptation practical for most compilations. The long-term goal of our research is to discover effective techniques for deriving compilation sequences that are efficient enough for routine use.

Advances are needed on at least two fronts to make these techniques practical: better methods for evaluating the impact of a specific compilation sequence and more effective techniques for searching the space of possible sequences. Because the search algorithm must evaluate each sequence that it considers, techniques to efficiently evaluate the results of compilation have the potential to drastically reduce the overall cost of finding good sequences. In particular, accurate performance models may drastically reduce the time spent in evaluation [24]. Better search techniques can also decrease the number of sequences that must be evaluated. This work focuses on understanding the properties of these search spaces and designing search algorithms that capitalize on those properties.

Unfortunately, we know too little about the interactions between optimizations to reason about these spaces analytically. Existing models are not yet mature enough for our large-scale experiments [33]; thus, our approach is experimental. We conduct two kinds of experiments: enumerations and explorations. Enumerations examine a subset of the optimizations and evaluate each point in that subspace for a single program and a sample input. They help us understand properties of the search space and develop intuitions about how search algorithms should work. Explorations use a specific search algorithm to find good sequences for a variety of programs under some objective function. They let us evaluate the effectiveness of a search technique across a variety of programs and confirm the intuitions developed in the enumerations.

Our Prototype Compiler We perform these experiments using our prototype adaptive compiler. As shown in Figure 1, it has front ends for Fortran and C, a set of 16 distinct optimizations, and back ends that generate code for a sim-

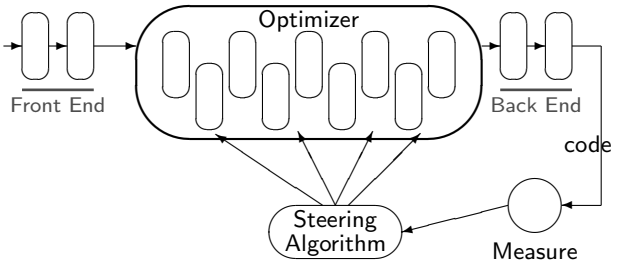

Figure 1: Prototype Adaptive Compiler

ulator and for the SPARC. The optimizations can be run in arbitrary order. The prototype uses a feedback loop and a steering algorithm to pick a compilation order, measure its impact, and adjust the compilation order. We have run the prototype compiler with objective functions that measure operations executed, static code size, and a property believed to correlate with power consumption. Throughout the paper, we refer to the individual optimizations by the following code letters:

- `c` *Sparse conditional constant propagation*; this optimistic global algorithm combines constant propagation with unreachable code elimination [30].
- `d` *Dead code elimination*; this global algorithm implements the SSA-based algorithm due to Cytron et al. [13], with an improvement due to Shillner [11].
- `g` *Optimistic global value numbering*; an application of Hopcroft's DFA-minimization algorithm to global redundancy elimination [1].
- `l` *Partial redundancy elimination* (PRE); this global technique uses data-flow analysis to find both full and partial redundancies and eliminate them [26].
- `m` *Renaming*; this global pass builds a name space suitable for the implementations of PRE and `lcm`. The compiler inserts it (automatically) before PRE or LCM. It can also run as a standalone pass.
- `n` *Useless control-flow elimination*; a simple, global algorithm for eliminating useless nodes and edges the control-flow graph [11].
- `o` *Logical peephole optimization*; this peephole optimizer examines operations that are connected by def-use chains rather than proximity in the instruction stream [14].
- `p` *Iteration peeling*; this pass peels the first iteration from each inner loop that it finds.
- `r` *Algebraic reassociation*; this global transformation uses associativity and distributivity to reorder expressions [4].
- `s` *Register coalescing*; this global pass combines live ranges that are connected by copy operations. It uses the infrastructure of a graph coloring register allocator [6].
- `t` *Operator strength reduction*; this pass implements an SSA-based global algorithm [9].
- `u` *Local value numbering*; the classic algorithm, credited to Balke, finds local redundancies and folds local constants.
- `v` *SCC-based value numbering*; This global, optimistic, hash-based algorithm performs value numbering and constant folding [28].
- `x` *Dominator-based value numbering*; this algorithm operates over regions in the dominator tree, giving it a different scope that `v` [5].
- `y` *Value-numbering over extended blocks*; this pass extends Balke's algorithm (`u`) to extended basic blocks [5].
- `z` *Lazy code motion* (LCM); this improvement to PRE provides more careful placement of inserted operations [22].

Enumeration Results The search spaces are huge. Schielke's experiments operated in a 10-of-10 space (sequences of length 10 drawn from 10 optimizations) with 10^{10} possible sequences. To keep the enumerations manageable, we used 10-of-5 subspaces (9,765,625 points) and small programs. We compiled each program with each sequence and recorded the corresponding fitness value. This process creates large but manageable datasets that we can analyze to gain insights into the structure of these spaces. (We confirm the insights with explorations of larger spaces for larger programs.) We also use the datasets in offline experiments, since lookup is faster than compiling the code and evaluating it.

Our first enumeration used a small Fortran program, `fmin`,[2] and optimized it to minimize operations executed. We chose the 5 optimizations experimentally; we started a hill-climber from 100 randomly-chosen points in a 10-of-13 space. We selected for use in the enumeration the 5 optimizations that occurred most often in the 100 final sequences found by the hill climber. They are loop peeling (`p`), partial redundancy elimination (`l`), peephole optimization (`o`), register coalescing (`s`), and useless control-flow elimination (`n`). We call this subspace `fmin+plosn`.

The `fmin+plosn` enumeration used 14 CPU-months on 3 processors; it took 6 wall-time months. Subsequent engineering improvements in the compiler have radically decreased the overall time for such enumerations; we can now enumerate a 10-of-5 space on 10 processors in about 2 weeks.

The results in `fmin+plosn` range from 1,002 to 1,716 instructions executed—a difference of 42%. The unoptimized code also executes 1,716 instructions. For comparison, the GA has found solutions at 822 (in a 10-of-13 space) and 835 (in a 10-of-16 space). The reader is cautioned against reading any significance into the fact that the GA found a worse solution in the larger space; the GA only probes a small fraction of the space.

The `fmin+plosn` enumeration shows that the search space is neither smooth nor convex. (Other enumerations confirm this result.) The `fmin+plosn` space has 189 *strict* local minima—points with the property that every string at Hamming-distance one has a higher fitness value.[3] If we define a *nonstrict* local minimum as a point where all Hamming-1 points have equal or higher fitness values, then `fmin+plosn` contains 31,995 nonstrict local minima.

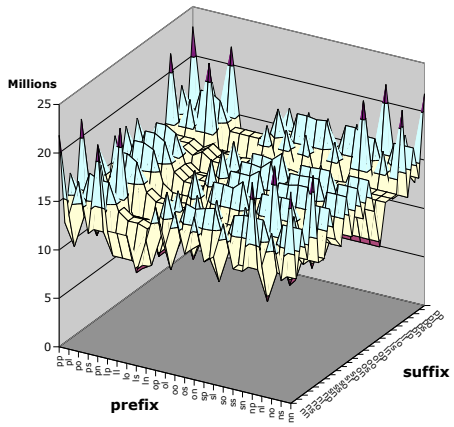

Figure 2: Strings of Length Four in adpcm+plosn

[2] `fmin` has 150 lines of Fortran source code organized into 44 basic blocks. It minimizes an external function using a combination of golden section search and successive parabolic interpretation.

[3] The Hamming distance between two strings is the number of positions in which they differ. Two strings are Hamming-1 if they differ in a single position.

Understanding the data from these enumerations is difficult because we cannot visualize a 10-dimensional space. The largest space that we can conveniently plot is a 4-of-5 space—strings of length 4 drawn from a set of five optimizations. The plot in Figure 2 shows the 625 fitness values for `adpcm-coder` from MediaBench in the 4-of-5 `plosn` space. The fitness values in this plot (and throughout this paper) give the number of instructions executed by the compiled code on a simulated RISC architecture. (We discuss this issue more in Section 4.) The surface resembles the surface of a heavily-cracked glacier; it conveys little or no intuition about the relationships between sequences and fitness values. The `fmin+plosn` space displays similar disorder, as shown in Figure 3.

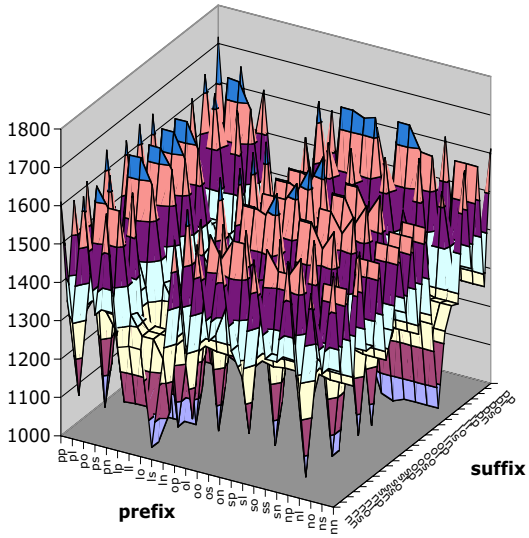

Figure 3: Strings of Length Four in fmin+plosn

To complicate matters, any visualization requires that we impose an order on `p`, `l`, `o`, `s`, and `n`. Figures 2 and 3 present the data with the strings ordered in the sequence `p`, `l`, `o`, `s`, and `n`. We would like to find an order for the axes where the data displayed properties that simplified the search. For example, the following interpretation of the 4-of-5 `adpcm` data would lend itself to a simple descent method.

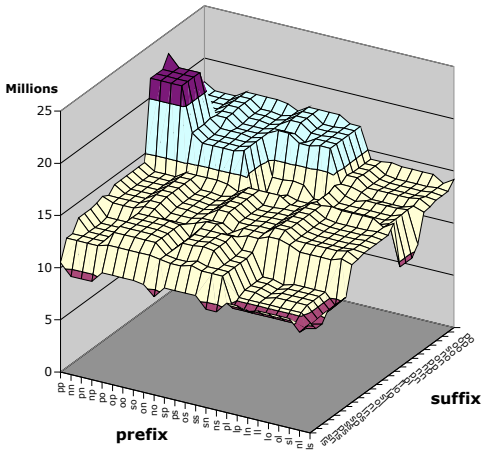

Figure 4: Reordered adpcm+plosn data

Unfortunately, this data does not correspond to any consistent ordering of the strings. (The two axes are labelled in different orders.) It was produced by averaging the fitness values across rows and sorting by row, then averaging fitness values by column and sorting by column. The 120 consistent orders for the data resemble Figure 2 much more closely than Figure 4. The same holds true of the other datasets that we have examined.

Measured Properties of the Search Space To date, we have enumerated five 10-of-5 subspaces using two small programs and optimizing for operations executed. The data demonstrate the range of possible improvement and the impact that selection of optimizations can have on the results. (In the fitness values for `adpcm`, the notation M indicates millions of operations.)

	Best Value	#	Worst Value	#	Range
`fmin+plosn`	1,002	1	1,716	1	41%
`fmin+pdxnt`	1,216	8	1,716	1024	29%
`zeroin+plosn`	832	3,099	1,446	1	42%
`zeroin+pdxnt`	1,020	8	1,446	1024	29%
`adpcm+plosn`	9.34 M	291	22.93 M	1	45%

The columns labeled # show the number of sequences that produce the given value (out of 9,765,625). The enumerations expose several properties of the search spaces that may be useful in the design of searches.

- The subspaces have many local minima, both good and bad. In `fmin+plosn`, most local minima lie within 5% of the best. However, as Figure 5 below shows, a second group of local minima lies 23% to 25% above the global minimum. Thus, any single hill-climber run might reach a "bad" local minimum, but the best result from multiple hill-climber runs (starting at random points) should find a "good" local minimum.

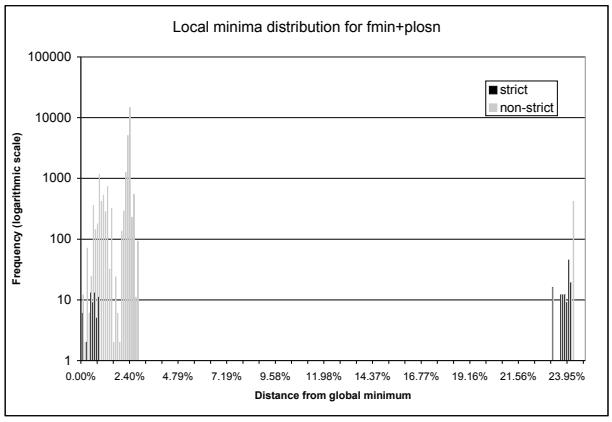

Figure 5: Distribution of Minima in fmin+plosn

- The distance from a randomly-chosen point to a local minimum, measured in Hamming-1 steps, is small relative to the size of the space. As shown in Figure 6 below, an impatient hill-climber finds a minimum in sixteen or fewer steps in the 10-of-5 spaces. Exploration studies in the 10-of-16 spaces confirm that this property appears to hold there, as well.

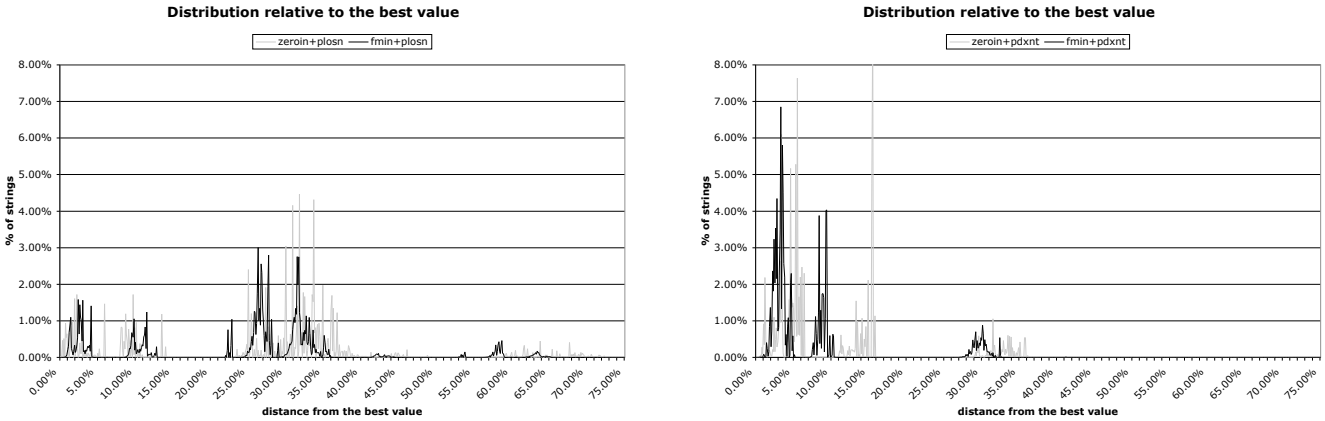

Figure 7: Distribution of Function Values in Different Spaces

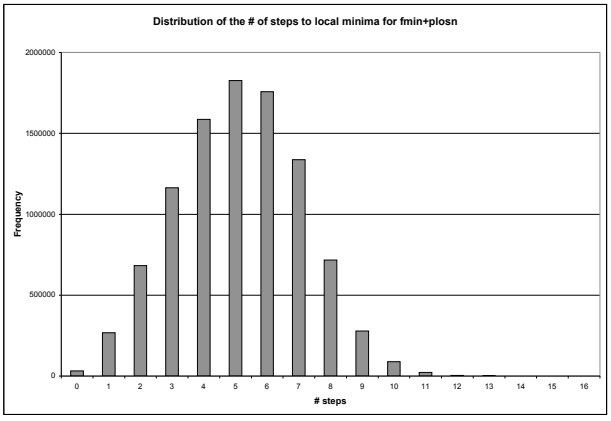

Figure 6: Hill-climber Steps to Local Minimum

- To differentiate between a good minimum and a bad minimum, we would like to build a reasonably accurate model of the distribution of fitness values. In the 10-of-5 subspaces, it appears that 1,000 probes produces a model that is accurate enough to let the compiler evaluate solution quality elsewhere in the space, as shown in Figure 8 below. The horizontal axis shows fitness value while the vertical axis indicates the percentage of the dataset at that value. (The model also provides a concrete upper bound on the results—the best probe.)

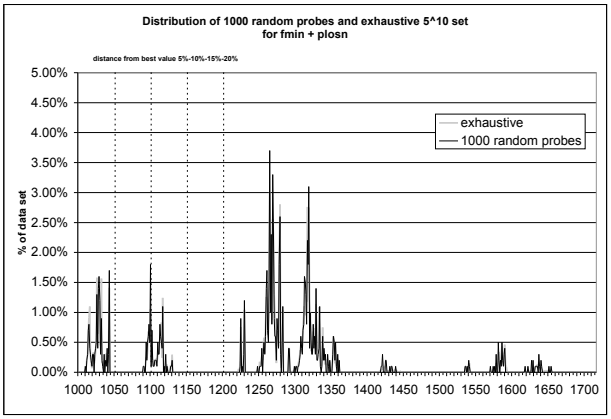

Figure 8: Random probes versus enumeration

- The distributions of function values and local minima depend on both the program and the set of transformations being considered. Compare the distribution of values for `fmin` and `zeroin` in two distinct spaces, `plosn` and `pdxnt`, as shown in Figure 7.

One further aspect of the subspace enumerations bears mention. The `pdxnt` subspace was chosen to address the perceived weaknesses of the `plosn` subspace. (The best sequence in `fmin+plosn` is 18% slower than the best 10-pass sequence found by a GA; that GA-selected sequence uses 8 distinct transformations.) In fact, the best sequence in the hand-picked `pdxnt` space is worse than the best sequence in `plosn`, by roughly 20%, for both `fmin` and `zeroin`. Choosing the best optimizations for a program is difficult; adaptive search is usually more effective than human insight.

The enumeration studies, while time consuming, have been limited to small programs and 10-of-5 spaces. From these studies we have drawn conclusions about the number and distribution of local minima. We believe that those conclusions apply to larger programs and larger search spaces. As support for this belief, we cite two distinct arguments. First, many other complex problems, such as 3-satisfiability, display similar structure; local search algorithms have proven effective in finding good solutions for those problems [27]. Second, the algorithms described in this section are designed to capitalize on these properties. The success of these algorithms and the fact that they behave as predicted are indirect evidence suggesting that the structural properties observed in the 10-of-5 spaces carry over to the 10-of-16 spaces.

4. SEARCHING THE SPACE

The enumeration studies provide insight into the structure of the search space. In particular, they show that the spaces have enough local minima that biased sampling techniques, such as multiple hill-climber runs, should find good solutions. This is fortunate, *since enumeration is not practical for realistic spaces.* This section describes several search techniques and presents the results of using those search techniques to find sequences for a collection of benchmark programs. These exploration studies help us evaluate the suitability of specific search techniques for use in a sequence-finding adaptive compiler. In this section, we report results from exploration studies in the 10-of-16 space.

	GC-10		GC-50		HC-10		HC-50		GA-50	Sequence	Testing
fmin	88.3%	588	88.3%	588	74.0%	413	73.9%	2,124	73.5%	pppxocdlsn	75.4%
zeroin	71.2%	1,377	70.5%	5,307	74.0%	462	71.0%	2,054	69.6%	oplvscdzsn	72.5%
adpcm-c	68.0%	838	68.0%	1,687	70.4%	402	68.5%	2,238	67.0%	prppocvdsn	67.6%
adpcm-d	70.0%	1,567	70.0%	3,573	70.0%	423	70.0%	2,328	69.4%	prpopcdlsn	70.1%
g721-e	84.1%	303	84.1%	303	85.3%	502	83.3%	2,857	81.3%	pnpppcdzsn	81.5%
g721-d	83.9%	303	83.9%	303	82.2%	427	82.1%	2,752	80.8%	vpppppcdsn	80.5%
fpppp	81.1%	678	81.1%	1,078	78.8%	632	75.3%	3,220	75.3%	nopclvdsnp	76.9%
nsieve	57.5%	1,859	57.5%	6,521	57.5%	391	57.5%	2,452	57.5%	pppppcdzsn	53.2%
tomcatv	120.3%	303	120.3%	303	85.4%	458	83.0%	2,555	82.7%	crxpotvdsn	82.8%
svd	77.6%	545	77.6%	728	77.1%	631	75.8%	2,756	73.9%	cztpvodvsn	70.2%

Figure 9: Performance in the 10-of-16 Space Relative to the Fixed Sequence "rvzcodtvzcod"

While we run explorations in the larger 10-of-16 space, we also run them in the enumerated 10-of-5 spaces. The results of the enumerations play an important role in the exploration experiments. Having precomputed data makes explorations of the 10-of-5 subspaces fast—lookup costs less than compilation followed by evaluation. In the enumerated spaces, we can also evaluate the quality of solutions found by an exploration—because we have complete data, we know the global minimizer(s).

Improving the Genetic Algorithm The GA used in our LCTES 99 paper used a population of 20 sequences in a 10-of-10 search. Sequences were ranked by fitness value. At each generation, the GA removed the worst string along with 3 others randomly chosen from the lower half of the population. Replacements were generated by random choice from the top half of the population and single-point crossover. All strings, except the most fit, were then subject to mutation. (Exempting the best string ensures its survival.) The GA found its best sequences in 200 to 300 generations.

Extensive experiments with variations on the GA improved its behavior. Our best results use a population of 50 to 100 sequences and single-point random crossover with fitness-proportionate selection. (In choosing sequences for use in the crossover operation, each sequence is assigned a weight proportional to its fitness value, normalized by the best value yet seen.) At each generation, the best 10% of the sequences survive without change. The rest of the new generation is created by repeatedly picking a pair of sequences and performing the crossover operation followed by a low probability, character-by-character mutation. If a new sequence has already appeared in an earlier generation, it is mutated until an untried sequence is found. (Viewing the GA as a search, it is wasteful to explore the same sequence twice. The GA maintains a hash table of sequences and their fitness values to detect duplicates.) With these modifications, the GA finds its best solutions within 30 to 50 generations.

Hill Climbers An alternate search strategy is to choose an initial point, evaluate its neighbors, and move to a point with a better fitness value. Such "hill climbers" can be efficient and effective ways to explore a discrete space. Because simple hill climbers stop at local minima, they are often invoked multiple times from distinct starting points. (The enumerations showed that the search spaces are littered with local minima.) The resulting algorithm performs a stochastic descent with multiple restarts from randomized points.

Our hill climbers define the "neighbors" of a string x as all Hamming-1 strings for x. To find the step that produces the greatest change in fitness value, the search must evaluate each Hamming-1 neighbor; in the 10-of-5 subspaces, each string has 40 Hamming-1 neighbors (4 other options in each of 10 positions). The enumeration results suggest that a hill climber should typically halt, in `fmin+plosn`, in 8 or fewer steps, with an upper bound of 16 (see Figure 6). Multiple trials of a hill climber should find good results quickly.

To confirm that hill climbers work as well in the large spaces as they do in the enumerated spaces, we ran them in the 10-of-16 spaces. In practice, a random descent algorithm (first downhill step) often outperforms a steepest descent algorithm (largest downhill step). An *impatient* random descent algorithm shows particular promise; it bounds the number of neighbors that it evaluates at each step. (The underlying idea is similar to Baluja and Scott's *patience* parameter [2].) To compensate for its shallower local exploration, our impatient random descent algorithm performs more trials. As the results in Figure 9 show, the impatient random descent algorithm competes well against more expensive techniques.

Greedy Constructive Algorithms Greedy algorithms produce good results for many complex problems (e.g., list scheduling). To assess the potential of greedy techniques for our search spaces, we implemented and evaluated a greedy constructive algorithm. It views the search space as a DAG where the root node represents the empty sequence. In a space with n optimizations, the root node has n children, representing the sequences of length 1. Every other node has $2n$ children that represent the effects of prepending or appending a different optimization to the parent node's sequence. The constructive algorithm walks downward from the root, moving at each step to the child with the best fitness value, until it constructs a sequence of the desired length.

Ties in fitness value complicate the greedy search. We have built versions of the constructive algorithm that explore all equal-valued paths (GC-EXH), that repeat the search fifty times and break ties randomly (GC-50), and that use a breadth-first exploration (GC-BRE). Ties, when they occur, can drastically increase the cost of GC-EXH; for example, on the program `adpcm-d`, GC-EXH explores 936,222 sequences in the 10-of-16 space while GC-50 examines 3,202 and GC-BRE examines only 425 sequences.[4]

Predictive Algorithms Clearly, a transformation can only improve the program if the code contains instances of the inefficiency that the transformation attacks. The magnitude

[4]As expected, the more expensive searches can produce better results. In this case GC-50 does as well as GC-EXH. GC-BRE produces code that executes 1% more instructions.

	GC-10		GC-50		HC-10		HC-50		GA-50	Best Sequence	Testing
fmin	90.4%	557	90.4%	557	90.5%	557	90.4%	2,280	90.2%	pppvcppodp	96.2%
zeroin	94.2%	1,419	94.2%	2,584	93.8%	363	92.5%	1,941	91.1%	pppopsndyc	94.7%
adpcm-d	100.6%	1,317	100.5%	1,317	100.0%	339	100.0%	1,645	99.9%	rpppppopcdx	99.9%
svd	109.0%	463	108.9%	463	91.7%	513	91.7%	2,670	89.5%	cozdtppvdn	85.9%

Figure 10: Results on the SPARC architecture

of that improvement depends on both the transformation and the fraction of runtime attributable to that inefficiency. Eventually, we expect to find correlations between properties of the input program and good sequences. These correlations may lead to algorithms that predict good sequences by examining the source code. For example, loop-free code is unlikely to benefit from operator-strength reduction or loop peeling. At this point, we have not begun to relate program properties to optimization sequences.

Comparing the Algorithms The table in Figure 9 presents results of typical runs of several search algorithms. GC-10 and GC-50 use the greedy constructive method with random tie breaking; GC-10 performs 10 trials while GC-50 does 50 trials. HC-10 and HC-50 are impatient random descent hill climbers. They test up to 10% of the current point's neighbors, using 10 trials and 50 trials respectively. GA-50 is the genetic algorithm, run with a population of 50 sequences and 100 evolutionary steps.

The programs are a mix of benchmark codes from the suites that we use in regression testing the compiler. `fmin` and `zeroin` are small numerical routines from the Forsythe, Malcolm, and Moler book on numerical methods [15]. `svd` implements Golub and Reinsch's singular value decomposition [15]. `nsieve` is the classic sieve of Eratosthenes benchmark. `adpcm` and `g721` are taken from MediaBench, while `fpppp` and `tomcatv` are from Spec95.

For each program and each search algorithm, the table shows the best results from 3 runs of the search in the 10-of-16 space, using the transformations listed in Section 3. Each result consists of two numbers. The first is the number of instructions executed by the compiled code on a simulated RISC architecture.[5] Since we cannot know the best sequence in the 10-of-16 spaces, the table shows the number of executed instructions as a percentage improvement over the execution of code compiled with the fixed-sequence version of the compiler, which uses the sequence `rvzcodtvzcod`.[6] The second entry gives the number of sequences that the search evaluated. GA-50 always evaluates 4,550 sequences.

The "Sequence" column shows the best sequence found for that program. The "Testing" column shows the performance of the best sequence on an alternate input dataset. (Since `fpppp`, `nsieve` and `tomcatv` use no external data, we modified parameters and initializations to change the computation.) The data shows no systematic bias in favor of the training data. In fact, some of the testing datasets produce better results than the training data, presumably because the sequence targets code that matters more in the testing data than in the training data.

All the techniques find consistent improvements. The only negative results arise with the greedy constructive technique on `tomcatv`. GA-50 often finds marginally better results than the other techniques, but it finds them at a significantly higher cost in compile time.

The tables in Figures 9 and 10 show results for GA-50, a genetic algorithm with a population size of 50 run for 100 generations. In practice, the GA finds good sequences in fewer than 100 generations. The table in Figure 11 shows the results for `fmin` in the 10-of-16 space using three distinct genetic algorithms: 50×50 (50 sequences for 50 generations), 50×100 (50 sequences for 100 generations), and 100×100 (100 sequences for 100 generations). The data is presented in the same form as in Figure 9, so that the center column, 50×100, corresponds to the GA-50 column in Figure 9. While performing more work (either running the GA for more generations or using a larger population of sequences) can produce better results, the improvements are often marginal. Limiting GA-50 to 50 generations brings its cost down to 2,300 evaluations, making it cost-competitive with HC-50.

	50×50	50×100	100×100
fmin	73.5%	73.5%	73.5%
zeroin	69.6%	69.6%	69.6%
adpcm-c	67.0%	67.0%	67.0%
adpcm-d	69.4%	69.4%	69.4%
g721-e	81.3%	81.3%	80.8%
g721-d	81.5%	80.8%	80.5%
fpppp	75.9%	75.3%	75.3%
nsieve	57.5%	57.5%	57.5%
tomcatv	85.1%	82.7%	82.7%
svd	73.9%	73.9%	69.9%

Figure 11: Varying the Work Done in the GA

Evaluating a sequence involves compiling and executing the code. The enumeration studies took CPU-months for tiny programs. Using the insights from those large-scale experiments, we refined both our biased searches and our GA parameters to the point where we can work with codes from MediaBench and Spec. The results in this paper show that we can obtain good results with even fewer evaluations; for example, HC-10 in the 10-of-16 space found good solutions (relative to the GA) in 400 to 650 evaluations.

These results show that biased search can do nearly as well as the more expensive genetic algorithms. The improved efficiency of methods such as HC-10 and GC-10 will make it feasible for us to work with larger programs and validate these findings. We expect that this work will lead to practical systems that adaptively choose program-specific com-

[5]Using a simulated architecture allows us to run the experiments on a variety of underlying machines. The data in Figure 9 are taken from a large set of experiments that took several calendar months on a collection of machines including SPARCs running Solaris, MacIntosh G4 servers running Mac OSX, and a Pentium running Linux.

[6]The sequence `rvzcodtvzcod` was chosen by the compiler's original designers to optimize for speed. It represents a high level of optimization.

pilation orders. One of the next steps in this work is to replace actual execution with evaluation of a performance model. That work will let us validate our ideas on larger programs. (In a practical system, large codes will likely be optimized in smaller pieces, so the results from this paper will carry over to those larger programs.)

The experiments reported in Figure 9 target a simple simulated RISC architecture. Figure 10 shows results on the SPARC architecture for `fmin`, `zeroin`, `adpcm-d`, and `svd`, in the same format as Figure 9. In general, we see a smaller variation in performance with the Sparc back end than with the simulator. We are investigating the reasons for this effect. Our Sparc back end is new and the compiler may need some additional optimizations to take full advantage of it. The disappointing results for `adpcm-d` support that idea. `fmin` shows more sensitivity to training/testing effects on the Sparc than on the simulated architecture.

5. LONG-TERM VISION

The overriding goal of our work is to lay the foundation for a new generation of compilers—adaptive compilers that adjust their behavior to produce the best code that they can in any particular circumstance. These compilers will be self-steering and self-tuning. They will use multiple compilations in a feedback loop to discover good configurations for each combination of input program and target machine.

Our work on finding compilation sequences demonstrates that we can find good sequences using algorithms that only examine a tiny fraction of the possible compilation orders. Before these techniques are practical, however, many open problems must be solved.

Search Algorithms The experiments described in this paper have focused on determining properties of the search spaces in which the prototype adaptive compiler works. Knowing about the structure of these search spaces has helped us devise better search algorithms; the hill climbers, in particular, are susceptible to tuning based on properties of the underlying space. For example, HC-50 does surprisingly well by exploring less of the local space and using more random starting points. Search algorithms that explicitly consider the structure of the input program may produce better results than the algorithms that we have shown.

Effect Modeling Our enumeration studies have built an empirically-derived model of the search space in which the prototype operates. An alternate approach would construct models of the interactions between optimizations, in an attempt to predict the impact of sequences without applying them [33]. Composable interaction models, whether derived empirically or analytically, might lead to steering algorithms that model the space of transformation effects and explore it without needing to compile or execute the code.

Performance Modeling In optimizing for operations executed, evaluating the compiled code's performance is the dominant cost. Performance models that predict running time could greatly reduce the cost of evaluating a sequence. Because the expense is large, hybrid schemes that combine profiling with modeling might be practical. Improvements in this area have the potential to make adaptive, search-based compilation practical.

Learning Methods In the long term, we hope to develop learning methods that correlate source-language program characteristics with specific optimization strategies. If we can develop reasonably accurate techniques to predict good optimization strategies from properties of the source code, it could significantly reduce the number of evaluations required to find a good program-specific sequence.

Building Compilers The primary impediment to building adaptive compilers appears to be the fact that most compilers are not written in a modular style, with well defined interfaces and limited interpass communication. Further research is needed to discover how optimizations should be structured to improve the opportunities for command-line adaptive control.

6. ACKNOWLEDGEMENTS

The research described in this paper has been supported by National Science Foundation Grant CCR-0205303 and by the Department of Energy through the Los Alamos Computer Science Institute. Many people have contributed to this work, through their programming efforts, their insightful questions, and their support and encouragement. The people who have worked in the Scalar Compiler Group at Rice over the years have put together a system that uniquely positioned us to pursue this research. Randy Chow, Fredrica Darema, Jose Muñoz, Ken Kennedy, and Andy White have all encouraged us in this work. To all these people go our heartfelt thanks.

7. REFERENCES

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

[2] S. Baluja and S. Davies. Fast probabilistic modeling for combinatorial optimization. In *Proceedings of the 15^{th} International Conference on Artificial Intelligence and 10^{th} Innovative Applications of Artificial Intelligence Conference*, pages 469–476, Madison, WI, USA, July 1998. AAAI Press/MIT Press.

[3] D. Bernstein, D. A. Goldin, M. C. Golumbic, H. Krawczyk, Y. Mansour, I. Nashon, and R. Y. Pinter. Spill code minimization techniques for optimizing compilers. *SIGPLAN Notices*, 24(7):258–263, July 1989. In *Proceedings of the 1989 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[4] 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.*

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

[6] 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.

[7] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In

Proceedings of the 4^{th} Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4), Dec. 2001.

[8] 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.

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

[10] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21^{st} century. *Journal of Supercomputing*, 21(1):7–22, Aug. 2002.

[11] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan-Kaufmann Publishers, 2003.

[12] 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.

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

[14] 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.

[15] G. E. Forsythe, M. A. Malcolm, and C. B. Moler. *Computer Methods for Mathematical Computations*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1977.

[16] M. Frigo. A fast fourier transform compiler. *SIGPLAN Notices*, 34(5):169–180, May 1999. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[17] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 ICASSP Conference*, volume 3, pages 1381–1384, 1998.

[18] E. D. Granston and A. Holler. Automatic recommendation of compiler options. In *Proceedings of the 4^{th} Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, Dec. 2001.

[19] L. Johnsson. Private communication. Discussion regarding algorithm choice in the Thinking Machine numerical libraries., Oct. 2003.

[20] 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.

[21] T. Kisuki, P. M. Knijnenburg, M. F. O'Boyle, and H. Wijsho. Iterative compilation in program optimization. In *Proceedings of 8^{th} Workshop on Compilers for Parallel Computers, CPC 2000*, pages 35–44, Jan. 2000.

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

[23] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Tools, and Compilers for Embedded Systems*, pages 12–23, June 2003.

[24] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, New York, NY, USA, June 2004.

[25] D. Mirkovic and S. L. Johnsson. Automatic performance tuning in the UHFFT library. In *Proceedings of the Interational Conference on Computational Science*, May 2001.

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

[27] B. Selman and S. Kirkpatrick. Critical behavior in the computational cost of satisfiability testing. *Artificial Intelligence*, 81(1-2):273–295, 1996.

[28] L. T. Simpson. *Value-driven Redundancy Elimination*. PhD thesis, Rice University, May 1996.

[29] 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.

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

[31] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–25, 2001.

[32] 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.

[33] M. Zhao, B. Childers, and M. L. Soffa. Predicting the impact of optimizations for embedded systems. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Tools, and Compilers for Embedded Systems*, pages 1–11, June 2003.