Compilation Order Matters: Exploring the Structure of the Space of Compilation Sequences Using Randomized Search Algorithms†

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

Rice University
Houston, Texas, USA

15 November 2003

Overview

Most modern compilers operate by applying a fixed sequence of code optimizations, called a compilation sequence, to all programs. Compiler writers determine a small set of good, general-purpose, compilation sequences by extensive hand-tuning over particular benchmarks. The compilation sequence makes a significant difference in the quality of the generated code; in particular, we know that a single universal compilation sequence does not produce the best results over all programs [1, 2, 5, 6]. Three questions arise in customizing compilation sequences: (1) What is the incremental benefit of using a customized sequence instead of a universal sequence? (2) What is the average computational cost of constructing a customized sequence? (3) When does the benefit exceed the cost?

To answer these questions, we must develop a good understanding of how quality of the generated code varies with the choice of compilation sequence over the entire space of sequences for a given program. In particular, we need to know (1) What percentage of the set of possible compilation sequences falls within a specified neighborhood of the true optimum sequence? (2) How are these nearly optimal sequences distributed in the sequence space? Do good sequences cluster in particular regions of the space? Or are they distributed evenly? (3) Is the sequence space riddled with shallow local minima? Can random sampling in the space reliably achieve near-optimal solutions? More importantly, we need to know if there are structural properties shared by compilation sequence spaces for a broad range of programs. Knowledge of common structural features of the space of compilation sequences enables identification of algorithms that find nearly optimal sequences with high probability. The cost of building customized sequences can then be reliably estimated.

In this paper, we empirically investigate the space of compilation sequences for a variety of programs. These include `fmin` and `zeroin`, as well as several programs from the Media and Spec benchmarks. `fmin` and `zeroin` are small enough to permit exhaustive enumeration of an interesting subspace of compilation sequences. We analyze this exhaustive data to identify some important properties of the compilation sequence space. About 15% of the sequences are within 10% of the optimum and about 30% of the sequences are within 20% of the optimum. There are many local minima in the sequence space, and about 80% of the local minima fall within 20% of the true optimum for both `fmin` and `zeroin`. The minima are clustered, and a graph-theoretic analysis reveals that the clusters are highly non-uniform in size. These properties suggest that *descent algorithms with randomized restarts* will be effective in this space. Indeed, in the compilation subspace of `fmin` and `zeroin` with over 9 million sequences, descent with randomized restarts finds customized sequences that are between 1-5% of the optimum after examining an average of about 25-150 sequences.

We then test the generality of these observations by evaluating the performance of both descent with randomized restarts and genetic algorithms on programs from the Media and Spec benchmarks. For these programs we do not know the optimal compilation sequence, since it is not feasible to enumerate and evaluate all elements in the full sequence space. Therefore we compare the performance of a sequence against that of the universal sequence used by our compiler. Surprisingly, for the cost of evaluat-

† This work has been supported by the National Science Foundation through grant CCR-0205303 and by the Los Alamos Computer Science Institute. For more information on this project, see `http://www.cs.rice.edu/~keith/Adapt`

ing between 1500 to 9000 sequences, both randomized search algorithms find custom compilation sequences that beat the performance of the universal sequence by 10-30%. The same number of independent random probes fail to generate such improvements across the range of programs we studied. We believe that this failure is due to the fact that the probability of finding a good sequence in these large spaces is extremely small.

In sum, we present one of the first empirically derived cost-benefit tradeoff curves for custom compilation sequences. These curves are for two randomized sampling algorithms: descent with randomized restarts and genetic algorithms. They demonstrate the dominance of these two methods over simple random sampling in sequence spaces where the probability of finding a good sequence is very low. Further, these curves allow compilers to decide whether custom sequence generation is worthwhile, by explicitly relating the computational effort required to obtain a program-specific sequence to the incremental improvement in quality of code generated by that sequence.

1 Custom compilation sequences

Compilers operate by applying a fixed sequence of code optimizations, called a compilation sequence, to all programs. Currently, most effective compilers include ten to twenty optimizations, drawn from the hundreds that have been proposed in the literature. To quote [4]: "Compile-time program optimizations are similar to poetry: more are written than are actually published in commercial compilers". One of the many reasons that such optimizations are not included in a compiler's arsenal is that they tend to be special-purpose optimizations that only help a certain class of programs. Their inclusion in a fixed compilation sequence can hurt most other programs.

To design compilers that effectively use all available optimizations, we need efficient techniques for computing program-specific compilation sequences. However, picking the best compilation sequence for a specific program, target architecture, and a compilation objective is hard. There is little theoretical understanding of the effect of particular compilation sequences on the properties of the compiled code. There are complex uncharacterized interdependencies between choices at a given point in the sequence and choices upstream and downstream of it, e.g., the use of transform a may eliminate opportunities for transform b downstream, or create situations where the effect of a later transform c is enhanced.

Thus, to determine the quality of a sequence, we have to evaluate it by compiling the program on the specified architecture and measuring the properties of the resulting code. This is an expensive process. Any approach that relies on enumeration or exhaustive search of the combinatorial sequence space is unlikely to be practical for real programs. Today, most compilers offer a small number of universal compilation sequences (`-O1`, `-O2`, `-O3`, ...) discovered manually by designers. Universal compilation sequences do not produce the best results over all programs and inputs [1, 2, 5, 6, 3]. If given universal sequences are not a good fit to the program or the architecture, today's user has little recourse.

With the ready availability of compute cycles, we believe that it is time to investigate whether it economically feasible for a compiler to calculate a customized sequence. Before we identify specific methods for computing customized sequences, it is important to obtain a ¡quantitative understanding of the relationship between the quality of a sequence and the computational effort needed to find it. We know that finding *optimal* compilation sequences for programs has worst case complexity exponential in the number of optimizations. We do not know theoretical bounds on the difficulty of finding approximately optimal solutions, i.e., sequences whose performance is within a specified neighborhood of the optimum. If we know the distribution and prevalence of approximately optimal solutions in the space of sequences, we can make the cost/benefit tradeoff for customization.

Thus, our goal is to obtain an understanding of the structure of the sequence space for particular programs, architectures and compilation objectives. We characterize structure of the sequence space by determining the following properties.

1. What percentage of the set of possible compilation sequences falls within a specified neighborhood of the true optimum sequence? If, say, the performance of *half* of the sequences, fall within 10% of the true optimum, then on average, *two* random probes of the space will suffice to get a good custom sequence, if good sequences are distributed uniformly. This percentage (i.e., half) characterizes the effectiveness of repeated, independent random sampling of the sequence space. In Section 2 we derive the mathematical relationship between the percentage of "good" sequences and the number of independent random probes of the sequence space.

2. How are these nearly optimal sequences distributed in the sequence space? Do good se-

quences cluster in particular regions of the space? Or are they distributed evenly? If good sequences are distributed evenly, simple random sampling might turn out to be the most effective strategy for searching the space of sequences.

3. Is the sequence space riddled with shallow local minima? This property has implications for the design of search algorithms in the sequence space as well as the computational effort needed to find approximately optimal solutions.

Since theoretical characterizations of the sequence space still elude us, we adopt an empirical approach to determining its structure. We collect data on the performance of *every* sequence in a large subspace (size 5^{10}) of the set of all possible sequences for two programs on two target architectures. It is important to note that this is not the approach we propose for actually calculating good custom sequences. This is a surveying mission to map out the landscape of the sequence space. Our goal is to learn enough about the space from this surveying effort to develop effective search algorithms for **other** programs, architectures and compilation objectives. Clearly, our success depends on how "typical" our initial exploration sites are, and how much commonality there is among sequence spaces for the unsurveyed programs, architectures and compilation objectives.

Our experimental plan has two phases. In the first phase, we select two interesting programs of moderate complexity, a set of five optimizations, and explore the subspace of all compilation sequences of a given length (10)[1] drawn from the five chosen optimizations. These five optimizations are `p` (peeling one iteration of a loop), `l` (partial redundancy elimination), `o` (peephole optimization over logical windows), `s` (register coalescing via graph coloring), and `n` (dead code elimination). These five were the top transforms found among sequences discovered by a constrained exploration of the full space of optimizations. Our exploratory programs are two floating-point computations, `fmin` and `zeroin`. These two programs are small enough to permit enumeration of the space of 5^{10} sequences, from which we can obtain a characterization of the landscape of solutions.

We analyze the data obtained from the exhaustive survey of `fmin` and `zeroin` to develop a clear picture of the distribution of good sequences in the space. As detailed in Section 2, for both programs about 15% of the sequences are within 10% of the optimum and about 30% of the sequences are within 20% of the optimum. There are many local minima in the sequence space, and about 80% of the local minima fall within 10% of the true optimum. A graph-theoretic analysis of the space of good sequences reveals that are many small isolated clusters of good sequences and a much larger single cluster for solutions less than 2.6% of the optimum. The isolated clusters disappear and solutions that are within $x > 2.6\%$ of the optimum coalesce into one giant cluster. These properties suggest that *descent algorithms with randomized restarts* will be effective when we require solutions that are within 2.6% of the optimum. Multiple retries from many randomly chosen starting points can reach the isolated clusters. When the solution clusters coalesce, simple random sampling is very likely to find good sequences. Indeed, in the compilation subspace of `fmin` and `zeroin` with over 9 million sequences, descent with randomized restarts finds customized sequences that are between 1-5% of the optimum after examining an average of about 25-150 sequences. Because of the way in which the five generating optimizations were chosen, this space has a much higher percentage of good sequences in the space. This property is verified by the fact that about 30-300 independent random sampling probes yields sequences that are within 1-5% of optimum for `fmin`. On larger benchmarks, and over the full space of transformations, we expect the fraction of good sequences to be much lower and simple random sampling techniques to fare much worse.

We test this conjecture by evaluating the performance of both descent with randomized restarts and genetic algorithms on `fmin`, `zeroin` and `svd`, as well as programs from the Media and Spec benchmarks on the full set of optimizations. For these programs we do not know the optimal compilation sequence, since it is not feasible to enumerate and evaluate 13^9 elements in the full sequence space[2]. Therefore we compare the performance of a sequence against that of the universal sequence used by our compiler. Surprisingly, after evaluating between 1500 and 9000 sequences, both randomized search algorithms find custom compilation sequences that beat the performance of the universal sequence by 15-30%. 3000 independent random probes are needed to generate similar improvements across the range of programs we studied, confirming our belief that the probability of finding a good sequence in these large spaces is extremely small.

The rest of the paper is organized as follows. In Section 2, we present the results of our initial exploratory surveys on `fmin` and `zeroin` on the limited

[1]We chose length 10 based on the length of the universal sequence in our compiler.

[2]We use 13 optimizations in these experiments, and consider compilation sequences of length 9.

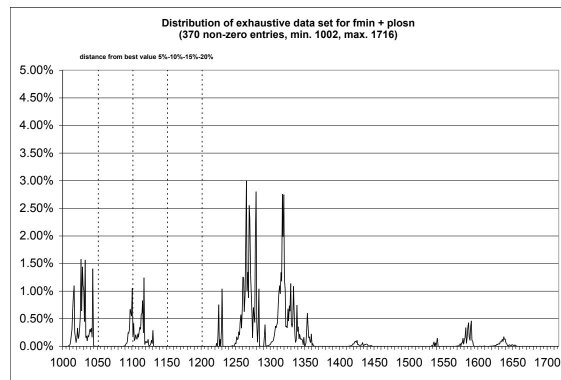
subspace. In the next section, we describe our results for `fmin`, `zeroin`, `svd` and programs from the Media and Spec benchmarks on the full compilation sequence space. In Section 4, we present one of the first empirically derived cost-benefit tradeoff curves for customized sequences. These curves are for two randomized sampling algorithms: descent with randomized restarts and genetic algorithms. They demonstrate the dominance of these two methods over simple random sampling in sequence spaces where the probability of finding a good sequence is very low. These curves allow compilers to decide whether custom sequence generation is worthwhile, by explicitly relating the computational effort required to obtain a program-specific sequence to the incremental improvement in quality of code generated by that sequence.

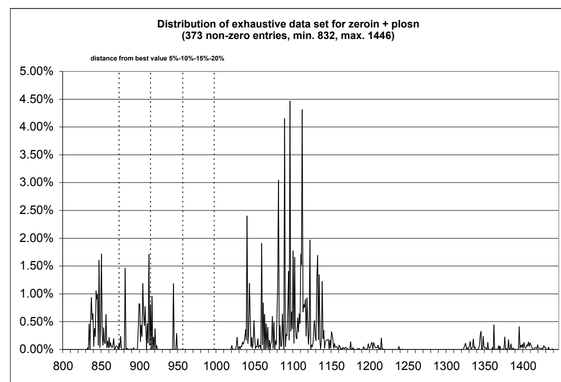
2 Surveying the compilation sequence landscape

For our initial survey experiments, we chose `fmin` and `zeroin`, two Fortran programs about 150 lines in size with around 40 basic blocks. `fmin` computes the minimum of a unimodal function by a combination of golden section search and parabolic interpolation. `zeroin` searches for the zero of an input function between given bounds. Both these functions are part of several commonly used mathematical libraries.

We selected five optimizations out of the complete set available to our compiler and enumerated all sequences of length 10 constructed from the five transformations. The selection procedure was as follows: we ran a descent algorithm with 100 randomly chosen starting points on the full sequence space for these two programs. This process is very fast, requiring on average about 50 sequence evaluations per restart. We picked the top five optimizations that occurred in the best sequences we found. The compilation sequence length was limited to ten because the universal strings in our compiler contain 9 or 10 optimizations. The compilation objective in this experiment is the minimization of dynamic operation counts.

Collecting data from this enumerative experiment was challenging, requiring 5^{10} or $9,765,625$ compilations. Fortunately, the optimization passes on our compiler are completely reorderable; this is one of the primary requirements of compilers that can support such enumerative analyses. Our first attempts consumed 14 CPU-months and six months of physical time. We made several improvements to our data gathering process, and we can now perform the `fmin` and `zeroin` enumeration over the `plosn` subspace in about 10 days. Of course, additional hardware can reduce this time even further, because the enumeration computation is embarrassingly parallel.





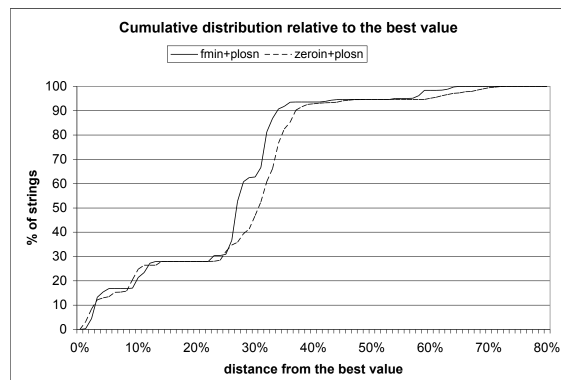


Figure 1: The distribution of dynamic operation counts for `fmin` (top) and `zeroin` (center) in the `plosn` subspace. The plot at the bottom shows the percentage of sequences that are within $x\%$ of the optimal solution for `fmin` and `zeroin`.

Figure 1 shows the distribution of dynamic operation counts for the 5^{10} compilation sequences of both `fmin` and `zeroin`. Note how discrete the dis-

tributions are, with large gaps in the dynamic operation counts. For instance, there are no sequences with operation counts that are between 14-20% of the optimum. The best sequence for `fmin` is 41.6% better than the empty compilation sequence (1002 vs 1716 operations), and the best sequence for `zeroin` is 42.5% better (832 vs 1446 operations). No sequence in the `plosn` subspace does worse than the empty compilation sequence. This, however, is not the case on the full sequence space, where it is easy to get results that are worse by more than a factor of two over the unoptimized program. The graph at the bottom of Figure 1 shows that about 15% of the sequences are within 10% of the optimum, while nearly 30% of the sequences are within 20% of the optimum. These statistics suggest that good sequences are not uncommon in the `plosn` subspace.

If these solutions are distributed randomly, we can expect repeated random sampling of the space to yield good customized sequences. Let a good sequence be one whose performance is within x% of the optimum, and p be the probability that a random sequence in the `plosn` subspace is good. The probability of picking a good sequence at the Nth probe, after failing the previous $N-1$ times is $p(1-p)^{N-1}$. If ϵ is a bound on the probability of failure to obtain a good sequence after N independent random probes, we have

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

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

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

The number of samples is not a function of the size of the underlying space, it only depends on the percentage p of good solutions in the space, and the probability $1-\epsilon$ of not missing a good solution. For `fmin` and `zeroin`, if we want solutions in the `plosn` subspace that are within 10% of the optimum, with probability greater than 0.999, we need no more than 44 samples. This is borne out by our random sampling experiments.

The previous bound is computed under the assumption that good sequences are uniformly distributed in the space. But how exactly are they distributed in the `plosn` subspace of `fmin` and `zeroin`? We need to empirically estimate the distribution. The estimation requires us to define a neighborhood relation among sequences. We say that a sequence is a neighbor of another if the Hamming distance between them is one, i.e., they differ in exactly one position. There are other possible definitions of neighborhood that remain to be investigated. All the results presented in this paper use the Hamming distance of one definition of neighbor.

We construct a graph whose nodes are sequences that lie within x% of the optimum. We draw an edge between two sequences in this graph that are Hamming distance one apart. Clusters are connected components of this graph. Figure 2 shows the number and sizes of the clusters in the solution space as a function of x. For $x < 2.6\%$, there are several isolated clusters (as many as 42, most of them of size 1), and a single large cluster. This is illustrated in the bottom plot of Figure 2. For $x < 2.6\%$, sequences in the large cluster are on average at a Hamming distance of 6.5 from the smaller isolated clusters. The smaller clusters are within Hamming distance 2.8 of each other. A dramatic transition occurs in the space of solutions, after $x > 2.6\%$. All solutions coalesce into one cluster, and there is a sharp increase in the number of good sequences. We believe there is a phase transition in the average case complexity of finding a good sequence in the `plosn` subspace. This transition occurs around $x = 2.6\%$. The implication of this finding is that for both `fmin` and `zeroin`, the task of finding a sequence that is within 2.6% of the optimum requires qualitatively different methods than those for finding sequences over 2.6% of the optimum. For the "hard" region, algorithms with multiple restarts will be important to ensure that isolated clusters can be reached. For the "easy" region, simple random sampling might suffice, because there are so many sequences satisfying the goodness criterion, scattered in the sequence space.

While cluster analysis gives us an overall view of how good solutions are spread through the space, it does not tell us how easy it is to reach them from different starting points in the space. Is there a natural slope to the terrain which can guide a descent algorithm to a good solution? How widespread are local minima in the space? How many of these local minima are "good" (i.e., within x% of the optimum)? For `fmin`, there are 31,995 local minima, of which 189 are strict. Below $x = 2.0$, less than 13% of the local minima are good. Between $x = 2.0$ and $x = 2.6$ the number of good local minima rises sharply from 13% to 80%. For $x > 2.6\%$, 80% of the local minima are good. This finding suggests a phase transition in the complexity of the problem around $x = 2.6$. Below it, few local minima are good; above it, most local minima are good. A simple descent algorithm will be particularly effective for $x > 2.6\%$, and descent algorithms with multiple randomized restarts will be

necessary for $x < 2.6\%$ to overcome the preponderance of poor local minima.

To determine whether are there natural gradients in the space, we measure the average number of steps taken by descent algorithms. Figure 4 documents the number of steps taken by a descent algorithm with 50 randomized restarts. The algorithm generates neighbors randomly and makes a downhill move as soon as a neighbor with a lower value is generated. The algorithm examines at most 10% of the neighbors. If it is unable to find a "better" neighbor after examining 10% of the neighbors, it stops and declares the sequence to be a local minimum. The characteristics of the descent algorithm are chosen based on the properties of the space we have determined thus far. To avoid getting trapped in poor local minima, especially for regions in $x < 2.5\%$ we need multiple restarts. The choice of 50 for this parameter was made empirically as the best tradeoff between the quality of final solution and the computational effort needed. For $x > 2.5\%$ far fewer restarts are needed. The number of neighbors for a sequence in this space is 40 (4 choices for each of 10 positions in a sequence). A greedy descent algorithm needs to examine all 40 before deciding on the next step. Figure 4 shows the distribution of the number of better neighbors for all 5^{10} sequences. It also shows the probability of missing a better neighbor if we examine only 10% of the neighbors (i.e., 4 sequences) at each step. Again, the choice of the 10% termination criterion is a tradeoff between cost of evaluation of sequences and the loss due to premature termination of a descent run. Interestingly, the average number of steps taken by this randomized local search algorithm for both `fmin` and `zeroin` is 1.9. The peak of the steps distribution in Figure 4 occurs at zero! This suggests that the space is pock marked with local minima and that our descent algorithm lands in one at the very start, 25% of the time. There are very few long descent runs in the space; the algorithm stops descending after less than two steps on average. A visualization of the much smaller space of sequences of length 4 (of size 5^4) is shown in the top plot of Figure 3. It confirms the findings of our descent algorithm in the larger space: (1) the presence of many local minima, and (2) the lack of long descent runs in the space. The lower plot in Figure 3 is an ordered view of the space of all sequences of length 4. The sequences are sorted according to dynamic operation count, and the axis labels reflect the sort order. However, the axis labels reflect no consistent ordering on `plosn`. The sorted view does shows that deep local minima are surrounded by shallow minima. Therefore search algorithms need mechanisms to escape these poor local

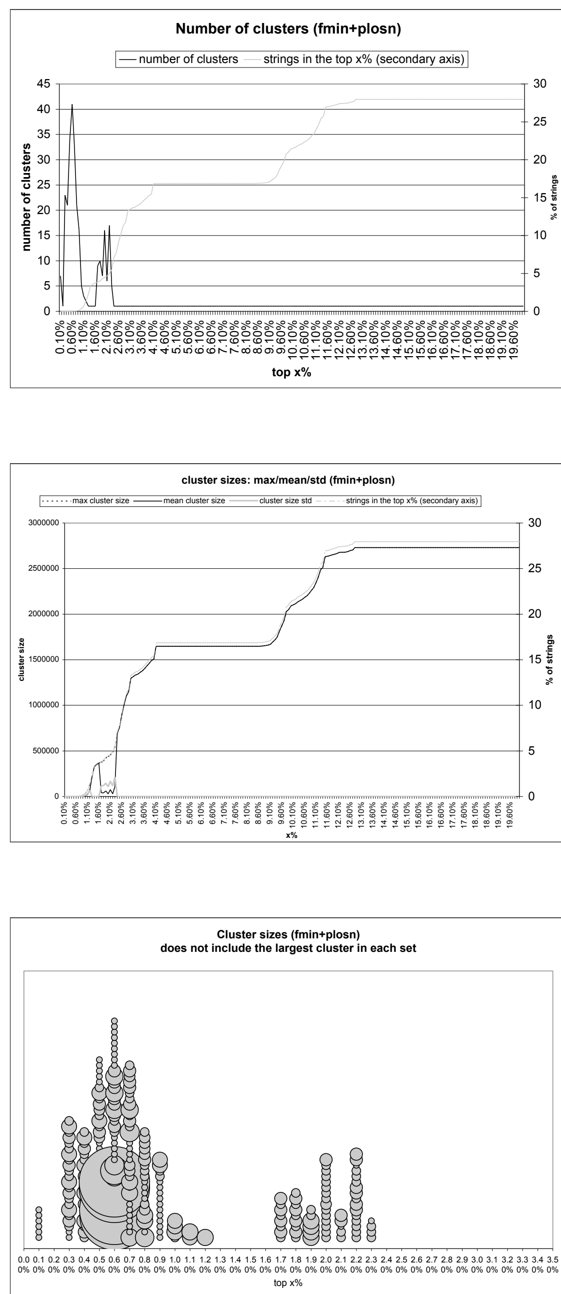
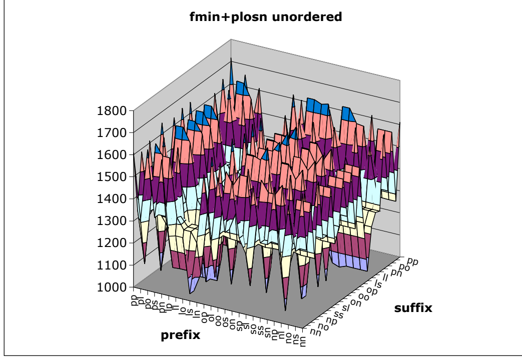


Figure 2: The top plot shows the number of clusters in the solution space of `fmin` as a function of the quality of the solutions. Solution quality is measured in terms of percentage from the optimal solution. The plot in the center shows the variation in cluster sizes as a function of solution quality. The bottom plot is a scale representation of how clusters other than the single giant cluster in the space, vary with solution quality. All the isolated clusters vanish at $x = 2.6\%$. We believe this is a phase transition in average case complexity for this problem. The plots for `zeroin` are very similar.



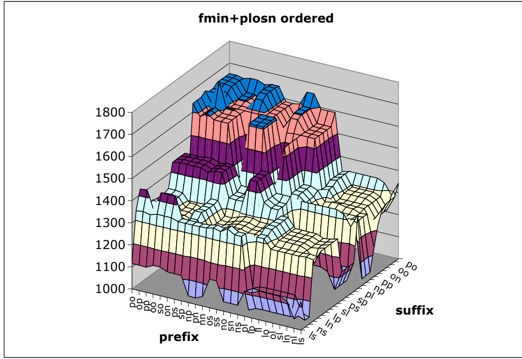
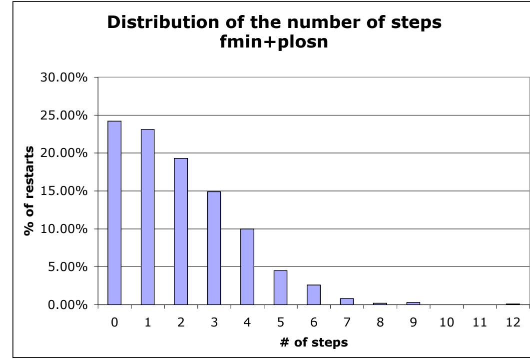


Figure 3: A visualization of the dynamic operation counts associated with all sequences of length 4 in the `plosn` subspace for `fmin`.



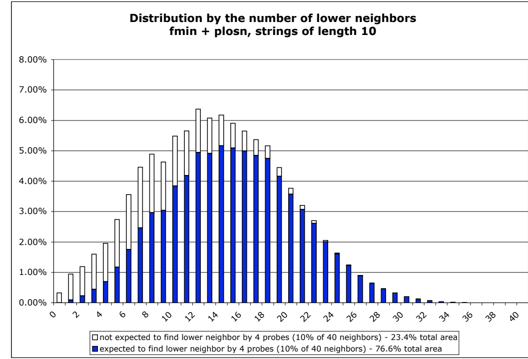
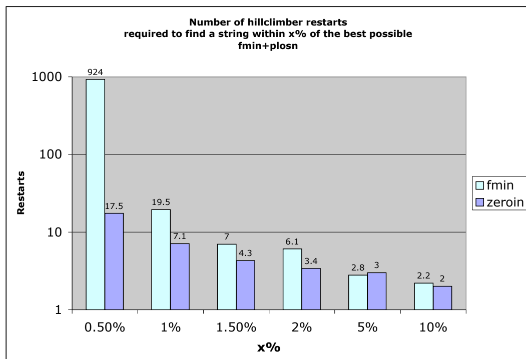


Figure 4: The behavior of descent algorithms for `fmin`.

minima.

Our survey of the landscape of the `plosn` subspace for `fmin` and `zeroin` offers one of the first computational glimpses of the space of compilation sequences. A predictive analytical theory of the impact of a sequence on these two programs has to capture the shape of the surface shown in Figure 3. Our empirical analysis shows that there are many local minima in the space, and that randomized restarts can overcome shallow local minima. The terrain is very rough, and most descent runs are short. Starting points matter for local search algorithms; deep local minima are reachable only if the starting points are very close to those minima. There is a sharp phase transition in this space. The problem of finding solutions that are within 2.6% of the optimum is qualitatively harder than the problem of finding solutions that are greater than 2.6% of the optimum. These properties need to be taken into account in the design of effective search algorithms for this space.

We conjecture that a descent algorithm with randomized, bounded neighbor selection (at most 10% of neighbors examined per step) with multiple randomized restarts is a cost- effective algorithm for finding good sequences in the `plosn` subspace for `fmin` and `zeroin`. Since descent runs are about two steps long and each step evaluates no more than 4 neighbors, we expect each run of our descent algorithm to use 8 sequence evaluations. The total cost is then $8 * R$, where R is the number of randomized restarts for the algorithm. Figure 5 shows the results of the performance of this algorithm against a repeated independent random sampling algorithm for `fmin` and `zeroin` on the `plosn` subspace. The top plot of Figure 5 shows the dramatic drop in the number of needed restarts as x exceeds 2.6. Recall that $x = 2.6$ represents solutions which are within 2.6% of the true optimum. Between 2.8 to 6.1 restarts suffice, compared to 924 for $x = 0.5\%$. This data reaffirms the phase transition phenomenon we noted earlier. The bottom plot of Figure 5 shows that randomized descent outperforms simple random probing for $x < 1\%$. Beyond $x = 1\%$, the number of good solutions in the space is large enough to make descent not worthwhile. We expect this phenomenon to generalize to other sequence spaces and programs. When a sequence space is rich in high quality solutions for



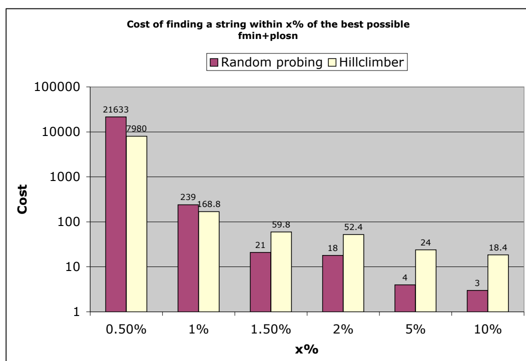


Figure 5: A comparative analysis of randomized descent with restarts against repeated independent random sampling for `fmin` and `zeroin` on the `plosn` subspace.

a particular program, simple random probing is the most cost-effective alternative. When the density of good solutions falls below a critical threshold, randomized descent is the algorithm of choice for computing customized sequences.

3 Exploring the full sequence space using randomized search

The complete mapping of the `plosn` subspace for `fmin` and `zeroin` is useful only to the extent that it provides algorithmic insights for effectively searching the full compilation sequence space for other programs. We conjecture that the general characteristics of these smaller spaces are universal. In particular, we expect the full space to have many many more shallow local minima. We also expect the probability of a random sequence being very close to optimal to be much much lower. We believe there will be phase transitions in the complexity of finding solutions within a particular percentage of the optimal; but the lack of knowledge of true optima make this analysis difficult for the present.

	Benchmark	Src	# of proc.	Src lines	ILOC lines	# of blocks
adpcm-c	Media	C	1	192	479	37
adpcm-d	Media	C	1	168	424	30
g271-c	Media	C	16	965	4066	268
tomcatv	Spec	For	1	192	2599	78
svd	fmm	For	1	351	2493	185
zeroin	fmm	For	1	125	332	39
fmin	fmm	For	1	160	434	59

Table 1: A list of the benchmark programs studied in this paper. The last three are implementations of numerical routines from Forsythe, Malcolm and Moler's classic text.

We attempt to verify these expectations by exploring the full space of optimization sequences for `fmin`, `zeroin`, `svd`, and some programs from the Media and Spec benchmarks. We use two different randomized local search algorithms: randomized descent with restarts as well as a genetic algorithm. The complete set of benchmarks in described in Table 1. There is considerable diversity in size and complexity in our test suite. For these programs, our comparison basis is the performance of the universal sequence used by our research compiler, since optimal sequences are unknown.

The descent algorithm uses 50 restarts and never examines more than 10% of its neighbors per descent step. The genetic algorithm we used was derived after some experimentation to find good parameter settings. We use populations of 50 and 100 sequences, a two-point random crossover, fitness proportional selection, and fitness scaling to exaggerate small improvements late in the search. We use 10% elitism, allowing 10% of the top sequences to survive without change for the succeeding generation. The mutation probability is 0.01. A unique feature of our GA is the mutate-until-unique operation. If the selection and crossover process yields a sequence that has already been generated in the current run of the GA, we mutate the sequence until a new one appears. This optimization cuts down wasteful exploration of the same sequence more than once in a run of the GA. Each GA runs for 100 generations.

Table 2 shows the results on `fmin` and `zeroin`. The dynamic operation count for `fmin` using the universal sequence is 1136, while that for `zeroin` is 978. Note that the optimal sequences in the `plosn` subspace for `fmin` and `zeroin` yielded counts of 1002 and 832 respectively. Both the descent algorithm and the GAs find custom sequences that are 25% better than the performance of the universal sequence for `fmin`

	fmin		
	dynops	cost	sequence
univ. seq	**1,136**	1	rvzcodtvzcod
GA 50	25.9%	4550	ozdpvsncd
GA 50	26.2%	4550	popppvdsn
GA 50	26.1%	4550	opvcdlsnd
GA 100	26.2%	9110	pvcopldsn
GA 100	26.1%	9110	polvcdscn
GA 100	26.2%	9110	ppvoclsdn
HC 50 10%	24.7%	1435	npclvosnd
HC 50 10%	25.7%	1355	ppcolsndn
HC 50 10%	25.6%	1315	popvgdzsn
R300	10.4%	300	pnsttvosn
R3000	23.9%	3000	smtzodlsn

	zeroin		
	dynops	cost	sequence
univ. seq	**978**	1	rvzcodtvzcod
GA 50	30.1%	4550	posvpscdn
GA 50	28.3%	4550	lospdpnzs
GA 50	30.1%	4550	popvdscnd
GA 100	30.1%	9110	ppvocdscn
GA 100	30.1%	9110	popvcdsnd
GA 100	30.3%	9110	ppnovcsdn
HC 50 10%	28.6%	1443	onppvsdnn
HC 50 10%	29.4%	1265	pnvlcosdn
HC 50 10%	27.7%	1145	pcosndzss
R300	14.8%	300	pnsttvosn
R3000	23.5%	3000	smtzodlsn

Table 2: The performance of our descent algorithm with 50 restarts (HC 50), GAs with population size 50 and 100, and independent random sampling with 300 and 3000 probes (R300 and R3000) for `fmin` and `zeroin` on the full compilation sequence space of size 9^{13}. Each run yields a different sequence suggesting the existence of isolated clusters of equivalent solutions. The Hamming distance between solutions is often as high as 8!

	adpcm-c		
	dynops	cost	sequence
univ. seq	**13,299,679**	1	rvzcodtvzcod
GA 50	32.0%	4550	ppnzpcnls
GA 50	32.0%	4550	pppnclnds
GA 50	32.4%	4550	ropcvspnd
GA 100	32.0%	9110	ppppcnlsn
GA 100	32.0%	9110	pnppcznds
GA 100	32.0%	9110	nppcldlns
HC 50 10%	32.0%	1435	pppcnllns
HC 50 10%	32.0%	1355	ppnncglns
HC 50 10%	30.9%	1315	dpnclnpms
R300	28.0%	300	pnsttvosn
R3000	30.9%	3000	nsszsncds

	adpcm-d		
	dynops	cost	sequence
univ. seq	**11,124,502**	1	rvzcodtvzcod
GA 50	30.6%	4550	rpvposcdn
GA 50	30.6%	4550	ropcvdpsn
GA 50	30.6%	4550	ropvpscnd
GA 100	30.6%	9110	prpocldsn
GA 100	30.6%	9110	rppvoscnd
GA 100	30.6%	9110	rppocvsnd
HC 50 10%	30.0%	1459	pnrzptcsn
HC 50 10%	28.4%	1171	ncdpplsps
HC 50 10%	30.6%	1459	przopcsnd
R300	27.0%	300	pnsttvosn
R3000	30.0%	3000	rgrvcgsnc

Table 3: The performance of our descent algorithm with 50 restarts (HC 50), GAs with population size 50 and 100 (3 runs each), and independent random sampling with 300 and 3000 probes (R300 and R3000) for `adpcm-coder` and `adpcm-decoder` on the full compilation sequence space of size 9^{13}. Each run yields a different sequence suggesting the existence of isolated clusters of equivalent solutions. The Hamming distance between solutions varies from 4 to 8.

(840 operations) and `zeroin` (684 operations). This comes at a cost of 4550 sequence evaluations for GAs with population size 50 and 9110 sequence evaluations for GAs with population size 100. In contrast, the descent algorithm with 50 restarts examines 1350 sequences on average for both these programs. For `fmin`, each descent step explores between 3.5 to 4.6 neighbors, and each run is between 4.1 to 5 steps long. There are longer descent runs in the full space, and the number of neighbors explored is on average almost twice that in the `plosn` subspace. Repeated random sampling with 300 probes yields sequences that are 10.4% better, while 3000 probes produce sequences that are almost as good as the ones found by the descent algorithm and the GAs. Note that the descent algorithm dominates repeated random sampling in this space, suggesting that the probability of finding a good solution at random is extremely low. There is considerable diversity in the sequences found across different runs of the same algorithm. This indicates that there are isolated clusters of equivalent solutions in the space, just as in the reduced `plosn` subspace for values for x under 2.6.

Table 3 shows results for the `adpcm` coder and decoder programs. The universal sequence produces dynamic operation counts of 13,299,679 and 11,124,502 respectively. Both the GAs and the descent algorithm find sequences that are between 30 to 32% better than our universal sequence. The cost for the descent search is about 1400 sequences, while the GAs have costs of 4550 (for GA 50) and 9110 (for GA 100). The Hamming distance between solutions on different runs is smaller for this benchmark, leading us to believe that the solutions that are within 30% of the universal sequence are in distributed among one or a few giant clusters. Further, the probability of a random sequence being as good as the universal sequence is higher in this space because 300 random

	g721-c		
	dynops	cost	sequence
univ. seq	**427,503,692**	1	rvzcodtvzcod
GA 50	15.8%	4550	npcpdznls
GA 50	15.2%	4550	pspnpcdls
GA 50	15.6%	4550	nczcppsnd
GA 100	15.7%	9110	npzppcnsd
GA 100	16.8%	9110	pcpdppvsn
GA 100	15.8%	9110	ncppznpds
HC 50 10%	15.3%	1134	pcnpldpns
HC 50 10%	14.3%	1309	cpncldncs
HC 50 10%	13.9%	1336	rzcopvdns

	g721-d		
	dynops	cost	sequence
univ. seq	**782,146,492**	1	rvzcodtvzcod
GA 50	16.62%	4550	ppcpdvpsn
HC 50 10%	14.9%	1827	pvpsncdgs
HC 50 10%	15.5%	1607	pppcnpvsd
HC 50 10%	14.1%	1669	pnpvcndps

Table 4: The performance of our descent algorithm with 50 restarts (HC 50), GAs with population size 50 and 100 for `g721-coder` and `g721-decoder` on the full compilation sequence space of size 9^{13}. We have not completed the experiments with 300 and 3000 independent random probes (R300 and R3000). Each run yields a different sequence suggesting the existence of isolated clusters of equivalent solutions. The Hamming distance between solutions varies between 6 to 8.

probes perform as well as the local search algorithms. It would be interesting to determine whether there are solutions that are more than 32% better than the universal sequence, and how they are distributed in the space

Table 4 shows our current results for the `g721` coder and decoder programs in the Media benchmark. The universal sequence produces dynamic operation counts of 427,503,692 and 782,146,492 respectively. The descent algorithms yield solutions that are between 14% to 16% better for the same costs as for all the previous benchmarks. The larger Hamming distance between equivalent solutions again suggests the presence of isolated clusters in the solution space. It also indicates that the probability of a random sequence being better than the universal sequence is very low. This conjecture will be vindicated when we complete our random probe experiments for this benchmark.

Table 5 shows the results we have thus far for the `tomcatv` program in the Spec benchmark. The universal sequence produces a dynamic operation count of 220,050,992. The genetic algorithm finds a giant cluster of solutions that are 15% better than the universal sequence. Our descent algorithm fails to find

	tomcatv		
	dynops	cost	sequence
univ. seq	**220,050,992**	1	rvzcodtvzcod
GA 50	15.0%	4550	rcpodvlsn
GA 50	15.0%	4550	rpcodpvsn
GA 50	15.0%	4550	rpcvosdpn
GA 100	15.0%	9110	rcvospncd
GA 100	15.0%	9110	rvcovdspn
HC 50 10%	12.0%	1776	oldtdvspn
HC 50 10%	10.0%	1798	orvozndps
HC 50 10%	13.0%	1727	roldczpsn

Table 5: The performance of our descent algorithm with 50 restarts (HC 50), GAs with population size 50 and 100 for `tomcatv` on the full compilation sequence space of size 9^{13}. We have not completed the experiments with 300 and 3000 independent random probes (R300 and R3000). Each run yields a different sequence suggesting the existence of isolated clusters of equivalent solutions. The Hamming distance between solutions varies between 4 to 8.

that cluster, and ends up in sequences located in isolated clusters whose performance is between 10-13% better than the universal sequence. The cost of the descent algorithm is about 1750 probes on average; while that of the GAs is 4550 (for GA 50) and 9110 (for GA 100). We have not completed the independent random probes experiments, but based on the results of the GA, we expect the 3000 probe experiment to find solutions as good as the ones the GA finds.

Table 6 shows the results for the `svd` program. The universal sequence produces a dynamic operation count of 5601. The descent algorithm finds solutions that are 28-30% better for an average cost of 1650 sequence evaluations. The Hamming distance between the solutions found by the three descent experiments suggests that there are isolated clusters of equivalent solutions in the space. For this benchmark GAs do a little better, yielding solutions that are 32-36% better than the universal sequence for costs of 4550 and 9110 evaluations, depending on the population size. Analysis of the sequences generated suggests that the GAs have found a very large cluster in the space (the Hamming distance between GA solutions is smaller).

Our randomized local search algorithms and independent random probes find solutions that are between 15-30% better than that produced by our universal sequence for a range of benchmarks. The performance of our local search algorithms reveals significant details about the distribution of good solutions in the full transformation space.

	svd		
	dynops	cost	sequence
univ. seq	**5601**	1	rvzcodtvzcod
GA 50	32.0%	4550	vodptvdsn
GA 50	36.0%	4550	odvtdpvsn
GA 50	36.0%	4550	odvtdpvsn
GA 100	32.0%	9110	vtpovdvsn
GA 100	33.0%	9110	covdtpsnd
GA 100	29.0%	9110	vtopscnds
HC 50 10%	30.0%	1622	otppvdsns
HC 50 10%	26.0%	1663	pnpvsonsd
HC 50 10%	28.0%	1763	popvdzsnp

Table 6: The performance of our descent algorithm with 50 restarts (HC 50), GAs with population size 50 and 100 for `svd` on the full compilation sequence space of size 9^{13}. We have not completed the experiments with 300 and 3000 independent random probes (R300 and R3000). Each run yields a different sequence suggesting the existence of isolated clusters of equivalent solutions. The Hamming distance between solutions varies between 4 to 8.

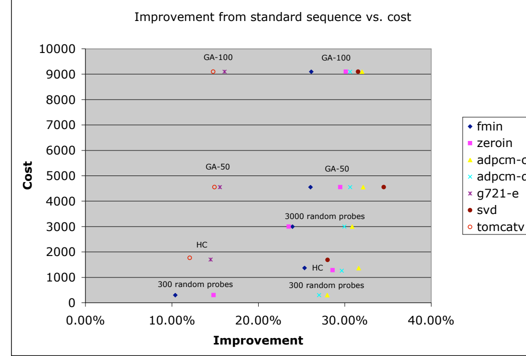


Figure 6: The cost-benefit tradeoff curve for two local search algorithms: randomized descent with restarts (HC) and genetic algorithms (GA) compared to independent random probing of the sequence space for the benchmarks studied in this paper. The cost-effective algorithms occupy the lower right hand corner of the figure.

4 The economics of custom sequence design

Figure 6 summarizes the discussion in the previous section and presents the cost/benefit analysis for custom sequence design. GAs with a population size of 100 do not appear to be cost-effective, they yield solutions that are similar in quality to GAs with population 50. For almost all programs, randomized descent with restarts strikes the right cost/benefit tradeoff as compared to GAs with population 50 and 3000 independent random probes. 300 independent random probes fails to yield solutions of sufficient quality for all but `adpcm` coder and decoder, which suggests that this set of programs has a sequence space rife with good solutions. For such spaces, the cost of descent is not worth the expected payoff. For all other programs, the probability of a random sequence being as good as the universal sequence appears extremely low, since randomized descent dominates both GAs and independent random sampling.

In conclusion, this paper presents empirical data on the structure of the sequence space for several interesting programs. Exhaustive enumeration of the `plosn` subspace yields insights into the difficulties in analytically characterizing the impact of sequences on particular programs. It suggests design criteria for search algorithms for the full sequence space for realistic benchmark programs. Our experimental results on the full sequence space indicate that lessons from the exploratory survey do have utility beyond the particular programs we investigated. We also show some of the first empirical economic tradeoff estimates to guide compilers in deciding whether or not to generate custom compilation sequences for specific programs.

Acknowledgements

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

References

[1] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *1999* ACM SIGPLAN *Workshop on Languages, Compilers, and Tools for Embedded Systems (*LCTES*)*, pages 1–9, May 1999.

[2] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. In *Proceedings of the 2001 LACSI Symposium*. Los Alamos Computer Science Institute, October 2001. Available at `http://www.cs.rice.edu/ keith/LACSI`.

[3] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek,

K. Gallivan, and D. Jones. Finding effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 12–23, June 2003.

[4] A. D. Robison. Impact of economics on compiler optimization. In *Proceedings of the ACM 2001 Java Grande Conference, Stanford University*, pages 1–10, 2001.

[5] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, 2003.

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